Palo Alto Research Center

Monitoring System Behavior In a Complex Computational Environment

by Mitchell L. Model

XEROX

Monitoring System Behavior In a Complex Computational Environment

by Mitchell L Model

CSL-79-1 JANUARY 1979

Abstract: See next page

This report reproduces a dissertation submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy. It is also available as Stanford University Computer Science Department Report CS-79-701.

CR Categories: 3.69, 4.42, 8.2.

Key words and phrases: Debugging, complex systems, Artificial Intelligence languages, graphics.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

© 1979

by

Mitchell L Model

Abstract

Complex programming environments such as the representation systems constructed in Artificial Intelligence research present new kinds of difficulties for their users. A major part of program development involves debugging, but in a complex environment, the traditional tools and techniques available for this task are inadequate. Not only do traditional tools address state and process elements at too low a conceptual level, but an Artificial Intelligence system typically imposes its own data and control structures on top of those of its implementation language, thereby evading the reach of traditional program-level debugging tools. This work is directed at the development of appropriate monitoring tools for complex systems, in particular, the representation systems of Artificial Intelligence research.

The first half of this work provides the foundation for the design approach put forth and demonstrated in the second. Certain facts concerning limitations on human information processing abilities which formed the background for much of the research are introduced. The nature of computer programs is discussed, and a concept of "computational behavior" defined. A thematic survey of traditional debugging tools is presented, followed by a summary of recent work. Observation of program behavior ("monitoring") is shown to be the main function of most debugging tools and techniques. Concluding this first part is an analysis of the particular difficulties involved in monitoring the behavior of programs in large and complex AI systems.

The second half presents an approach to the design of monitoring facilities for complex systems. The need for system-level tools similar to the ones traditionally available is indicated. A new concept called "meta-monitoring" replaces traditional dumps and traces with selective reporting of high-level information *about* computations. The importance of the visually-oriented analogical presentation of high-level information and the need to take into account differences between states and active processes are stressed. A generalized method for generating descriptions of system activity is developed. This method is based on a theoretical schematization of the fundamental structures and operations of computational systems and is easily instantiated for any particular AI system. Some specific display-based monitoring tools and techniques which were implemented for this work are exhibited. Several of the experimental monitoring facilities which were constructed in accordance with the principles of the proposed approach are described and their application to existing Artificial Intelligence Systems illustrated. While much of the research was performed in the context of the KRL-1 system developed at Xerox Palo Alto Research Center, the general applicability of the theory and techniques of the present work is demonstrated by one of these facilities, which acts as a monitor for MYCIN, a medical diagnosis system developed at Stanford University that embodies knowledge in the form of production rules.

Acknowledgements

The experimental aspects of this research would not have been possible (or even imaginable) without the resources -- human, software, and hardware -- of the Computer Science Laboratory at Xerox Palo Alto Research Center. I am most grateful for the generosity with which Xerox made these available to me, along with the financial support of a Student Research Assistantship.

Warren Teitelman contributed the indispensable foundation for all my experimentation during this project by making available to me his display-based version of INTERLISP and doing so long before its general release. Larry Masinter made important contributions to this work through his patient and imaginative technical assistance. Both had to endure endless confusion, requests, and pleas for help.

My thanks go to Edward Feigenbaum, Terry Winograd, and Cordell Green for serving on my reading committee and for their unwavering confidence in my "potential" in the face of what often appeared to be a prodigious lack of productivity. Without Edward Feigenbaum's encouragement I would have abandoned my graduate studies long ago. Moreover, earlier work with him gave me some of the initial ideas for this research. My adviser, Terry Winograd, has long been a source of intellectual stimulation and guidance for me. I am particularly happy to have had his support in the later stages of this work, when I needed it most. His thorough readings of various drafts and his constant assistance in reworking the text contributed greatly to the quality of the final product. Cordell Green's interest in my work increased my confidence in it, and it was through him that I maintained contact with research in Automatic Programming, an important factor in this effort. Daniel Bobrow and Gio Wiederhold also read drafts of this dissertation and suggested many improvements, for which I am grateful.

I thank David Levy for his encouragement, and for tolerating me as an officemate. His understanding and advice have helped guide my path.

I owe much to my parents, who made me what I am, and to those in my life who have helped me to understand and be what that is. Finally, and ultimately, I thank Janet, my wife, for her innumerable contributions to this dissertation, both substantive and stylistic; for her support and appreciation of my vision and my tortured ways of pursuing it; and for just about everything else.

Table of Contents

	Page
Acknowledgements	*v*
Table of Contents	*vi*
List of Figures	*viii*

Part One: Foundations

Chapter

1. Introduction	**3**
A. The Historical Neglect of Debugging	4
B. New Needs	8
C. The Design of Monitoring Facilities for AI Systems	11
2. Some Aspects of Human Information Processing	**15**
A. Human Memory	15
B. Limitations on the Capacity of Immediate Attention	17
3. Computer Programs	**23**
A. The Nature of Computer Programs	23
B. The Significance of High-Level Programming Constructs	27
C. Computational Behavior	30
4. Debugging Computer Programs	**37**
A. Detecting Programming Errors	37
B. Traditional Debugging Tools and Techniques	41
C. Recent Developments in Debugging Techniques	45
D. The Act of Debugging	52
5. Artificial Intelligence Systems	**57**
A. Difficulties in Defining the Behavior Desired of an AI System	57
B. The Organization of AI Systems	60
C. Data in an AI System	64
D. The Flow of Computation in an AI System	66

Part Two: The Design of Monitoring Facilities for AI Systems

Chapter		Page
6.	**An Approach to Monitoring the Activity of Complex Systems**	**73**
	A. System-Level Monitoring Tools	74
	B. The Power of Analogical Visual Presentations	78
	C. Discrete States and Dynamic Processes	83
7.	**Describing the Activity of a Complex System**	**87**
	A. The Fundamentals of Computational Systems	87
	B. Events	97
	C. The Uses of Events	101
8.	**Display-Based Tools for Monitoring Facilities**	**107**
	A. A Display-Oriented Programming Environment	107
	B. Graphical Tools for Monitoring Facilities	113
	C. Independent Monitoring	124
9.	**Some Experimental Monitoring Facilities**	**129**
	A. Viewing the Event Space of a KRL-1 Application System	131
	B. Dynamic Monitoring of System Activity in MYCIN	146
10.	**Conclusion**	**159**
	A. Summary	159
	B. An Unfulfilled Theme	161
	C. In Conclusion	165
Selected Bibliography		*167*

List of Figures

Figure		Page
1.	The Programming Language Hierarchy	27
2.	The Program Structure Hierarchy	27
3.	The Parallel Sequences of States and Operations	32
4.	Types of Programming Errors and their Detection	40
5.	The Domains of Program Code and Behavior	51
6.	The Relationship between the Mechanisms of Human and Artificial Intelligence	59
7.	The Extended Language Hierarchy	61
8.	The Extended Structure Hierarchy	61
9.	A Generalized Schematization of System Fundamentals	90
10.	The Fundamental Operations of Computational Systems	90
11a.	KRL-1 System Fundamentals	92
11b.	KRL-1 System Fundamentals (cont.)	93
12a.	INTERLISP System Fundamentals	94
12b.	INTERLISP System Fundamentals (cont.)	95
13.	A KRL-1 Unit with Triggers	96
14.	A MYCIN Production Rule	97
15.	A Typical DLISP Scene	111
16.	The Display and Printout of a Tree Structure	115
17.	The INTERLISP STORAGE Function and its Display as a Bargraph	117
18.	The Use of Bargraphs to Monitor Iterations	119
19.	INTERLISP Code Printed with Font Information	121
20.	The Display of a Simple KRL-1 Event Using Different Type Faces	122

Figure		Page
21.	A Representative Interrogation of the Event Space, Stage 1	133
22.	A Representative Interrogation of the Event Space, Stage 2	134
23.	A Representative Interrogation of the Event Space, Stage 3	135
24.	A Representative Interrogation of the Event Space, Stage 4	136
25.	A Representative Interrogation of the Event Space, Stage 5	137
26.	A Representative Interrogation of the Event Space, Stage 6	138
27.	A Representative Interrogation of the Event Space, Stage 7	139
28.	A Representative Interrogation of the Event Space, Stage 8	140
29.	A Representative Interrogation of the Event Space, Stage 9	141
30.	A Representative Interrogation of the Event Space, Stage 10	142
31.	A Representative Interrogation of the Event Space, Stage 11	143
32.	A Representative Interrogation of the Event Space, Stage 12	144
33.	A Representative Interrogation of the Event Space, Stage 13	145
34.	Concurrent Monitoring of a MYCIN Run, Stage 1	147
35.	Concurrent Monitoring of a MYCIN Run, Stage 2	148
36.	Concurrent Monitoring of a MYCIN Run, Stage 3	149
37.	Concurrent Monitoring of a MYCIN Run, Stage 4	150
38.	Concurrent Monitoring of a MYCIN Run, Stage 5	151
39.	Concurrent Monitoring of a MYCIN Run, Stage 6	152
40.	Concurrent Monitoring of a MYCIN Run, Stage 7	153
41.	Concurrent Monitoring of a MYCIN Run, Stage 8	154
42.	Concurrent Monitoring of a MYCIN Run, Stage 9	155
43.	Concurrent Monitoring of a MYCIN Run, Stage 10	156
44.	Concurrent Monitoring of a MYCIN Run, Stage 11	157
45.	Concurrent Monitoring of a MYCIN Run, Stage 12	158

Part One

Foundations

Chapter 1

Introduction

The purpose of the research reported on here was to explore the nature of debugging activity in a complex computational environment. Its specific goal was the development of appropriate monitoring tools and techniques for complex Artificial Intelligence systems. Familiarity with research in the field of Artificial Intelligence led to a particular focus on that domain, with the anticipation that most of the results would be relevant to other kinds of complex computational systems as well. Experience with Artificial Intelligence systems has shown that the observation and understanding of their behavior is frequently a difficult and frustrating task. The traditional approach to debugging is inadequate for dealing with the complexity of these environments. This work seeks to explain that inadequacy and demonstrate ways of satisfying the need which the user of an AI system has for information about the system's behavior.

This report is divided into two parts, the first establishing the foundations for the second. Although much of the material in the first part will be familiar to many readers, the particular perspective from which it is considered creates an important context for the theory and results put forth in the second part. The present chapter places this work in its historical context, defines the problems at which it was directed, and outlines the approach taken towards their solution. Chapter 2 discusses some limitations on human information processing capabilities that bear on the nature of computer programming and debugging. The third chapter considers the development of programming language constructs and the structure of computer programs as they relate to the psychological processes of the programmer. Also in that chapter, a particular conception of "computational behavior" is defined which is used in much of the later discussion. Chapter 4 contains an analysis of program errors. It reviews the techniques traditionally used in their detection and correction (*i.e.*, debugging) and describes some recent research in this domain. The chapter concludes with an analysis of debugging activity which shows that most debugging tools focus primarily on the observation of computational behavior. Chapter 5 considers the nature of Artificial Intelligence systems and the difficulties involved in observing their behavior using traditional techniques.

The second part of this work is concerned with the design of monitoring facilities for complex computational systems, specifically those in the field of Artificial Intelligence. Chapter 6 presents an approach to monitoring the behavior of complex systems which incorporates three new directions. First, it is argued, monitoring tools must address system-level structures and operations; furthermore, they must answer the user's questions *about* these instead of simply *showing* them to the user. Second, graphical presentation of analogic information is important for effective high-level communication. Third, the design of monitoring facilities must take into account the differences between discrete states and active processes, so that information can be presented dynamically where appropriate. Chapter 7 puts forth a method for producing descriptions of the activity of a computational system, based on that system's fundamental structures and operations. Chapter 8 describes some specific display-based techniques and tools that were developed for the implementation of monitoring facilities. In the course of the research, several experimental monitoring facilities were constructed utilizing the approach presented in Chapter 6, the activity description method of Chapter 7, and the display-based techniques and tools described in Chapter 8. The ninth chapter demonstrates the operation of two of these facilities, one designed for a trigger-based KRL-1 system and the other for MYCIN, a production rule system. The tenth and final chapter summarizes the work and suggests a direction for future research.

A. The Historical Neglect of Debugging

The need for techniques to aid in the analysis and correction of computer programs was apparent as soon as stored-program computers became a reality, and the typical programming environment offers several tools to aid the programmer in the detection of errors in his code. The history of programming technology shows that most debugging tools and techniques are variations on a small set of basic ideas that were discovered in the earliest days of computing. [These will be discussed in Chapter 4.] The past twenty years has seen little change in the nature of the debugging facilities available to the typical programmer. Debugging has rarely been the subject of theoretical investigation, and new ideas, techniques, and tools generally do not propagate beyond the environment in which they were produced. Despite the major role debugging plays in the program development process, it has simply not enjoyed the organized attention given throughout the development of computer technology to the construction of computers and the languages in which they are programmed. The problem is particularly acute with respect to high-level languages, as even now, a high-proportion of the available debugging tools were initially designed for use with machine- or assembly-language code.

Most of the few papers and books that do deal with debugging are introduced with expressions of wonder at this situation. In 1965, Halpern wrote that:

> Such is the nature of the epoch of computer usage we are moving into, armed with debugging tools inadequate even for today's problem -- children with Flit guns about to meet the army ants. There has been talk for years about "source-language debugging," and even some claims that it had been realized in this or that system; but the distance we have yet to go to make good this boast is sufficiently indicated by the frequency with which we have recourse to the core-memory dump. The dump, still the major debugging tool for all but formal errors in the use of source language, represents a reversion to the more primitive machine-oriented level that is due to bankruptcy on the application-oriented level." [Halpern, "Computer Programming: The Debugging Epoch Opens", p. 29. Although there is no indication given in the article, it seems likely that the reference to Flit is more than just symbolic: FLIT was one of the earliest in the influential series of debugging systems developed at Lincoln Laboratory and MIT in the late fifties and early sixties that produced the widely used DDT {Digital Equipment Corporation, *PDP-10 Reference Handbook.*}. RAID, a display version of DDT developed at the Stanford Artificial Intelligence Laboratory {Petit, *RAID*, 1970.}, is a further evolution in this entomocidal chain.]

Evans and Darley made the following comment in 1966:

> One consequence of recent interest in the development of large-scale time-sharing systems to provide on-line computer access to a large number of users has been the widespread realization that the usefulness of such a system is critically dependent on the quality of the software provided to facilitate the interaction between user and machine. In particular, one area of critical importance for effective utilization of such a system is that of facilities for program debugging. In view of the important role they play, surprisingly little attention has been paid to the development of facilities to aid in the process of on-line program debugging. Furthermore, much of the work in this field has been described only in unpublished reports or passed on through the oral tradition, rather than in the published literature." [Evans and Darley, "On-Line Debugging Techniques," p. 37.]

Gaines, in his 1969 study, found that debugging "has received very little detailed treatment in the computer science literature." He noted "that an examination of 17 introductory programming texts revealed only 4 in which the subject of debugging received more than a few brief sentences." [Gaines, *The Debugging of Computer Programs*, p. 2.] As late as 1973, Mann could state:

> Almost twenty years ago, higher-level languages began to free the programmer from several levels of detail and give him tools more amenable to his problem. . . . With this growth of language power there has not been a corresponding growth in debug tools. Thus the programmer must often revert to machine level to determine where his program was, and what data values were, at some point of interest. At best he has had to include additional statements in his program to create a partial record of its internal values and conditions. But they must be removed for the production version of the program and, worse, require recompilation to change them as clues appear and new viewpoints on the program's functioning are needed. . . . We find, then, a gap betwen programming language facilities for creating programs and tools for detecting and analyzing their performance." [Mann, "A Survey of Debug Systems," p. 183.]

Satterthwaite, writing in 1975, held an opinion similar to those of his predecessors:

> Software systems designed to aid in the debugging and analysis of computer programs have received relatively little systematic attention. Much interesting work is never openly published, either because it is embedded within proprietary packages, because it is considered incidental to the achievement of other goals, or because it is simply not thought worth documenting. As a result, many of the basic ideas and techniques are rediscovered regularly. In addition, accurate and unbiased surveys are difficult to prepare; they tend to reflect the surveyor's personal contacts, geographical location, and accidental discoveries. [Satterthwaite, *Source Language Debugging Tools*, p. 17.]

This general disregard of debugging in computer science constrasts starkly with the continual flow of new ideas and major theoretical and technological accomplishments produced in the related area of programming languages. Certainly this difference in emphasis and results cannot be justified on the basis of programmer needs. There is probably no sound way of estimating the extent to which a programmer spends computational and personal resources on debugging activity in the course of the average programming task, but there can be no disputing that it is large. One pair of authors, noting this contrast, writes: "The majority of commercial programmers spend as much as 50% of their time program testing, but all their formal training and development is concentrated in the other half of their jobs." [Brown and Sampson, *Program Debugging* p. ix.] Some might even find this estimate too low, especially for programmers working in a more research-oriented environment, where problems are perhaps less well understood. It is not just the theorists and system builders who neglect debugging: programmers themselves demean this part of their task. From the beginning, debugging has generally been considered an unfortunate nuisance which would be avoided if only programmers were more careful and had better languages and facilities to use. This widespread attitude is unreasonably idealistic in view of the continuing necessity for programmers to spend a major part of their time and effort on debugging, despite better training, better computational facilities, and better programming languages.

Why has there been so little organized attention paid to the development of debugging systems? Halpern advanced the interesting view that two earlier bottlenecks in the development of programming technology had to be overcome before debugging would become a problem. He wrote:

> Until sometime in the later '50's, the limiting factor for most applications was hardware. Much of a programmer's effort during those early years was spent on planning how to squeeze a useful program into 4K of core, and how to time the reading of the data so that it would be called for just as the drum brought it under the read head. The epoch of hardware-limited computing was brought to a close when machines in the class of the IBM 709, with its million-bit core, overlapped input/output, and 12 microsecond cycle time became common.
>
> With King Log deposed, King Stork arrived in the shape of software-limited computing. FORTRAN was still having teething troubles, COBOL was as yet but a gleam in the eye of the Dept. of Defense, and the rest was assembly-language coding and hand-punched binary patches. We are now nearing the end of this second epoch. FORTRAN is as reliable and taken-for-granted as the machine it resides in. COBOL has not only been implemented but has occasionally even been well implemented. There are special application languages for every human activity this side of truffle-hunting. We have, then, machines big and fast enough to execute most useful programs, and software varied and reliable enough to get those programs written; what now? Now: debugging. [Halpern, "Computer Programming: The Debugging Epoch Opens," 1965.]

Halpern's basic insight still seems a good one, despite there still being no widespread belief that debugging is the main obstacle in the programming process.

Evans and Darley pointed out another factor contributing to the particular scarcity of debugging facilities for high-level languages:

> . . . on-line debugging facilities for higher-level languages are in general less well-developed and less widely used (relative to the use of the languages) than their assembly-language counterparts. In part, this situation is probably a consequence of the wide diversity of languages in this class; probably it is still more a result of the fact that the small machines on which the assembly-language techniques originated and were cultivated were typically considered too small to support higher-level language compiling systems and were programmed almost exclusively in assembly language. Thus work in on-line debugging of higher-level languages is of comparatively recent origin." [Evans and Darley, "On-Line Debugging Techniques", p. 41.]

Another explanation was offered by Gaines:

> Computer programming is a creative art, best done by individuals, and debugging is the most highly individual aspect of programming. The techniques that may be useful for debugging depend very highly on the individual situation: the environment in which the programmer is working (computer, operating system, programming language, etc.), the debugging aids available, the particular algorithm being programmed, and the nature of the particular bug or bugs the programmer is trying to find. Furthermore, finding bugs for the most part requires an act of intelligence, i.e., a mechanical procedure can only occasionally indicate exactly what an error is. [Gaines, *The Debugging of Computer Programs*, pp. 1-2.]

To some extent, debugging has become a matter of programming practice, much like the extensive commenting of code. A comment of Knuth's should not surprise the experienced programmer:

> "The most effective debugging techniques seem to be those which are designed and built into the program itself -- many of today's best programmers will devote nearly half of their programs to facilitating the debugging process on the other half; the first half, which usually consists of fairly straightforward routines, will eventually be thrown away, but the net result is a surprising gain in productivity." [Knuth, *The Art of Computer Programming*, Vol. 1, p. 189.]

Thus, the programmer can code his own debugging tools directly into his programs, without having to rely on separate debugging facilities. Standard mechanisms used by programmers to obtain information about a program's activity include printouts of the values of important variables at strategic points and messages concerning control decisions. From this point of view, the value of *special* debugging systems is not necessarily that great, given the flexibility inherent in any language and its operating environment. This approach to debugging emphasizes its *ad hoc* aspects. As Brown and Sampson stated in the introduction to their book on debugging, "When we started work on that material the common reaction of people we spoke to was one of amazement: debugging is considered a kind of formless, esoteric art, only learnt by programmers through trial and error and almost impossible to discover techniques for it, let alone teach them." [Brown and Sampson, *Program Debugging*, p. ix.]

Another consideration that discourages development in this area is the large amount of communication that must occur between debugging facilities and their users. The programmer has already had to learn one or more languages and would be burdened by having to learn others simply for the purposes of debugging. Thus, even

where suitable facilities exist, the programmer may find that the potential value of the facilities is not worth the effort necessary to become adept at using them. Similarly, many debugging tools turn out to be too expensive in terms of computational resources to be generally useful. Efforts to provide debugging facilities that go beyond the few tools commonly supplied are often thwarted by these practical considerations. Moreover, those debugging facilities which are sufficiently simple and inexpensive to justify their use are not likely to be significantly more helpful to the programmer than mechanisms he can code directly into his programs.

Finally, programmers generally discount the importance of debugging within their overall task, as mentioned above. The fact is only reluctantly accepted that programs rarely perform as expected, regardless of how carefully they are written. A programmer enjoys and/or earns his living converting mental conceptions of computations into concrete representations. In contrast, getting the program to actually run is a significant nuisance, approached with impatience, always taking more time and effort than anticipated (often by orders of magnitude), and producing much frustration and little satisfaction. If the programmer has suitable underlings available, he may even assign them the task of debugging the program, sparing himself the trouble. Programmers are forced to become adept diagnosticians; yet, as long as available tools and techniques are *adequate* for overcoming the obstacles on the way to correct program performance, debugging will continue to be considered an uninteresting and irritating task. Without the impetus provided by an obvious *need* for more highly developed facilities, this situation is not likely to change. Vague desires for tools which are more convenient or more pleasant to use, but which still serve roughly the same purpose as existing ones, do not provide sufficient motivation for major developmental efforts. Grand conceptions of ever more powerful programming languages and machine architectures generate much enthusiasm, but due to the general belittlement of debugging activity, no such excitement is evoked within this domain.

B. New Needs

In recent years, developments in the nature of programming activity have rendered traditional debugging techniques obsolete in some contexts. The complexity of some modern languages and the programs written using them is significantly greater than that of the standard languages and programs of the previous two decades. A language such as SAIL is not merely a dialect of ALGOL, a mainstream computer language. By its addition of such facilities as associative storage and multi-processing capabilities, SAIL permits and encourages programming in a style very different from those resulting from the use of standard algorithmic languages. Likewise, PLANNER and

QA4 are not just LISP-oriented programming languages with a few extra features -- these languages invoke a fundamental concept of computation that is in principle significantly different from that of LISP. PLANNER, for instance, is based on assertions, theorems, and a backtracking-oriented control structure.

These new higher-order languages are used almost exclusively within the field of Artificial Intelligence. This should not be surprising. The *nature* of the computation performed using general purpose computers in commercial, military, administrative, and scientific contexts has not changed much since the first computers were put to work. Perhaps computers have been in use for too short a time for substantially new ways of acquiring and manipulating data to have arisen. Instead, most of the effort on the pragmatic side of computer science has been directed at making these computations and the equipment on which they depend faster, more reliable, more flexible, easier to program, less expensive, and smaller. [The demands of real-time systems, particularly in military applications, do represent a new direction for computation, but problems in that domain are the result of the demand for faster and more reliable performance rather than increased system complexity.] Artificial Intelligence has found a need for new approaches to symbolic manipulation. As that field emerged, it quickly became clear that the linear, mathematical, algorithmic orientation of traditional computation is not appropriate for the representation of processes involved in intelligent action.

The development of environments for Artificial Intelligence programming has not stopped at the new AI languages [*cf.* Bobrow and Raphael, *New Programming Languages for AI Research*, 1973.]. Representation languages are a further stage in the same direction. They surpass AI languages both in scope and in the sophistication of their facilities. Any programming language exerts pressure on its users to structure data and programs in certain ways. Representation languages make a greater commitment to the form that data and programs should take then do AI languages, just as AI languages make more of a commitment along these lines than do traditional algorithmic languages. In order to build up a hierarchy of system facilities, basic assumptions must be made which support the design of the entire system. Contemporary representation systems include those based on production rules, semantic nets, or some notion of "frame". Each one establishes a fundamental ontology which dictates the ways in which knowledge can be represented and manipulated.

Researchers in Artificial Intelligence consider themselves involved in the investigation of linguistic, psychological, and perhaps even philosophical issues. Their interests lie primarily outside the domain of engineering. Although advanced programming systems are crucial to the progress of their work, these researchers tend to see the unavailability of such systems more as a hinderance than as a major issue to be

addressed. Nevertheless, these workers do find themselves building the powerful representation and computation systems that are necessary for the progress of their research, because no one else will build these systems for them. As a direct consequence of the orientation of their builders towards theory and away from "mere" engineering, AI systems tend to be unreliable, constantly in flux, and lacking in all but minimal support facilities. There seems to be a prevalent attitude that these new systems are simply more-of-the-same traditional computational facilities. Through their failure to recognize the special technical needs of the users of complex programming systems, the builders of Artificial Intelligence systems create frustration and difficulty for those working in such environments, including themselves.

Many of the awkward aspects of AI programming environments are associated with debugging activity. Programming technology in this field has outgrown the traditional debugging tools and techniques that have served more or less adequately since the early days of computers. One of the purposes of AI and representation languages is to simplify the expression of intricate relationships among entities in the system and of complicated sequences of events. Another purpose is to free the user from the burden of managing the details of generally useful operations such as associative retrieval, backtracking, multi-processing, pattern matching, and logical deduction. To the extent that activity of greater complexity can be specified using the higher-level constructs of these new languages, the problems of obtaining information about the behavior of such systems are themselves of a higher order. Not only is it incrementally more difficult to monitor the activity of programs written in these new languages, but the information desired by the user is of a different nature.

Because AI and representation languages impose new kinds of control and data structures on top of those of the traditional languages in which they are implemented, the value of the standardly available debugging tools is greatly reduced. The AI programmer wants to probe the system at times and places reflecting the nature of the high-level control regime in terms of which he has structured his code, whereas traditional debugging facilities are oriented towards the control structure of the underlying implementation language. Information gained about the flow of computation and the state of data elements through the use of those facilities will often be at too low a level. This results in an overwhelming flood of details that obscures rather than illuminates the major features of the program's activity. Moreover, the user may not understand the implementation level details that would be obtained with traditional tools, as he may know little about the AI system's implementatation or about the underlying programming language.

Debugging tools designed to address high-level language structures would be more useful in these environments than the standardly available program-level ones. Nevertheless, development of these would only be a small step towards meeting the system user's needs. AI and representation languages provide mechanisms for organizing computations in complex ways. Use of these may lead to the distribution of control decisions throughout the system and to forms of multi-processing; these contrast with the essentially linear flow of computation in a traditional environment. Consequently, understanding the flow of computation in AI systems is a difficult task which requires assistance of a different *type* than that provided by traditional debugging tools. Traditional sorts of debugging dumps and traces are incompatible with the nature and quantity of the information about the execution of his code desired by the user of an AI system. New methods must be developed for the production and communication of monitoring information in complex environments such as these AI systems. This is the challenge addressed by the present work.

C. The Design of Monitoring Facilities for AI Systems

In the course of the research, the design and construction of monitoring facilities for AI systems was explored in large part through actual experimentation. Although attention was focused predominantly on KRL-1, a representation language being developed at Stanford University and Xerox Palo Alto Research Center [See Bobrow and Winograd, *An Overview of KRL, A Knowledge Representation Language.*], this work was intended to produce results that would be generally applicable to *any* AI or representation language. Towards this end, much of the experimentation was directed at developing a generalized approach to monitoring the behavior of AI systems. The second part of the present report discusses the results of this effort and illustrates some of the experimental tools and monitoring facilities that were constructed.

The approach developed for meeting the monitoring needs of AI system users is based on three themes:

- high-level orientation of monitoring facilities;
- graphical presentation of analogic information; and
- recognition of differences between discrete states and active processes.

The first step towards high-level monitoring facilties is accomplished by continuing a trend exhibited by some recent work in source language debugging tools. A monitoring facility for an AI system must address system-level constructs, not entities at the level of the programming language in which the system is implemented. Tools for examining the values of individual variables or data cells must be upgraded to show

system-level data structures, in appropriately formatted ways. Mechanisms for interrupting or altering the flow of computation are of little use unless the user can specify the interruptions or alterations in terms of the system-level operations and processes with which he is familiar.

A further step proposed in the direction of high-level monitoring tools is to replace the complete printout produced by traditional trace or dump facilities with a more interactive investigation of system entities. Data structures and processes in a robust AI system may have so much information associated with them that the user would not want to see it all, even if it were completely comprehensible. It is important that high-level monitoring facilities be able to answer questions *about* data structures and operations instead of simply showing them in full detail. It is also helpful for the system to compile low-level bits of information into higher-level structures that provide answers to the user's question at the level on which it was conceived. This concept is given the name meta-monitoring. A simple illustrative example that will be used again later involves iterations. The fundamental question a programmer has about the progress of an iteration is "how far along is it?". Details about the current value of the iterative index, or the start- and end-points of the iteration, are too primitive in a complex computational environment, and are likely to be more distracting than helpful. High-level monitoring facilities should be able to answer such a question based on knowledge about iterations and produce a response like "80%".

The second theme is based on a consideration of some facts about human physiology and psychology which reveal a strong visual bias in the human organism. These facts also show that sensory information is highly organized before it reaches the parts of the brain associated with abstraction, analysis, and other components of thought. The significance for monitoring facilities of these information processing characteristics of the human organism is that the pictorial, or analogical, presentation of information is often more effective than presentation in more abstract, symbolic modes. The monitoring information required by the AI system user is of such great extent and so complicated that visually-based aids are no longer the "frills" they have traditionally been considered. Devices such as bargraphs and tree-structure displays present analogic information which can be grasped "at a glance", whereas the user would expend much time and mental effort to understand the same information were it presented symbolically. Moreover, such display-oriented tools support the first component of the approach to monitoring, in that high-level information compiled from lower-order pieces can often best be shown analogically. To return to the iteration example, an effective monitoring device is a bargraph that "fills in" as an iteration progresses.

The third part of the approach to monitoring involves the recognition that to achieve an understanding of active processes, it is not sufficient to sequentially monitor individual discrete states. Traditional debugging tools are designed to provide information about the state of the computation at some particular time, leaving it up to the user to notice patterns which span the individual state descriptions. Analogical visual presentations are also useful in this context. They can make dynamic patterns much easier to notice and understand for the user. Moreover, there is a qualitative difference in the information that can be communicated through "movies" of active processes, as opposed to sequential "snapshots" of discrete states. Monitoring facilities designed with this distinction in mind offer novel kinds of aid to the user of a complex system.

Implementation of actual monitoring facilities based on the approach proposed in this work requires two further supporting components: (1) a mechanism for generating the raw information for the monitor to process; and (2) suitable graphics-based facilities for implementing the display-oriented tools. A proposal for the first of these is discussed in Chapter 7. It is based on a notion of "Events", which describe system activity. A generalized schematization of fundamental system structures and operations is exhibited. It is a straightforward matter to instantiate this scheme for some particular system by adding small amounts of Event generating code to the system where fundamental operations are performed on fundamental structures. Most systems have only a small number of fundamental structures. There are only a few basic operations that could apply to each of these. For each operation on each structure, there are a few facts that would in general be important to include in a description of the operation and its effects. Thus, adding the Event generation code involves only a minor effort, and the generation of the events during execution imposes only a small burden on the system. These Events can be used for several purposes, including:

- monitoring computational activity during system execution;
- replaying "movies" of system activity after execution, perhaps at varying speeds, levels of detail, or foci of attention;
- post-execution investigation of aspects of system activity; and
- reasoning within the system itself.

Chapter 8 discusses the second supporting component necessary for the general approach. The experimental monitoring systems constructed in the course of this research were implemented using a display-oriented version of INTERLISP developed by Warren Teitelman at Xerox Palo Alto Research Center. DLISP, as it is called, makes available facilities for high-level graphics programming within the context of a general-purpose programming language. Thus, DLISP frees the user from the burden of

having to manipulate display vectors or bitmaps, much as high-level programming languages free the programmer from having to deal with machine-level instructions and individual memory cells. One far-reaching affect which the general availablity of a system such as DLISP would have is that the implementation of the kind of tools described in Chapter 8 will be straightforward and undemanding. Then, display-based monitoring facilities can leave the luxurious realm of experimentation and become a practical reality.

Some of the particular features of DLISP that proved especially useful included multiple, overlapping windows; a pointing device (called a "mouse"); different styles, sizes, and faces for the display of text; and different background patterns for windows and arbitrary rectangular areas of the screen. Using these DLISP facilities, specific tools were constructed that would be useful in any AI system monitor. Graphical tree displays and various kinds of bargraphs proved especially valuable during experimentation. Multiple windows were used extensively to sort monitoring information into separate, coherent streams. Different font sizes, styles, and faces and the various background patterns available in DLISP provided a helpful way to highlight important information. The experimental monitoring facilities were designed to run separately from the systems for which they were designed. Chapter 8 also discusses methods for implementing this kind of independent monitoring and the advantages to be gained through its use.

Chapter 9 illustrates two of the experimental monitoring facilities. The first is directed at a trigger-based KRL-1 system for making travel arrangements. It allows the use to interactively interrogate the space of Events generated and recorded during system execution. The other facility is a monitor for the MYCIN system, an experimental medical diagnosis program developed at Stanford University. This second example illustrates the generality of the approach to monitoring, the description generation method, and the display-based tools. Very little effort was required to apply these to an environment other than that of KRL-1, the one in which they were developed. The separation of monitor and AI system proved valuable here. In this configuration, MYCIN is run on the SUMEX-AIM computer where it resides, while the monitor runs on the Xerox PARC computer, the two systems communicating with each other over the ARPANET.

Chapter 2

Some Aspects of Human Information Processing

The activities of computer programming and debugging involve a great deal of communication between the programmer and the computer system. Analysis of these activities must therefore take into account the information processing characteristics of both people and computers. Because the emphasis in this work is on the flow of information from the computer to the programmer, and because the communication features of computational systems are fairly straightforward and commonly understood, only the human side of these communication activities will be considered here. This chapter presents a characterization of some aspects of human information processing that are relevant to the programmer's efforts to understand computational behavior. The view developed here supports much of the material in the rest of this report and will be referred to throughout. Further discussion of human information processing is to be found in Section B of Chapter 6 below, which addresses human vision in particular.

A. Human Memory

The primary concern here is with restrictions on how much information a person can process at once that derive from certain quantitative limitations on human memory. Typically, modern psychological theories posit three or four levels of memory, labelling them something like Sensory Memory, Short-Term Memory, and Long-Term Memory, also including, perhaps, an Intermediate-Term Memory. [See Cofer, "An Historical Perspective," for a brief overview of the development of this view and some of its variations. Various researchers have attempted to construct a fully developed theory of memory. See, for example, Anderson and Bower, *Human Associative Memory*, 1973; Kintsch, *The Representation of Meaning in Memory*, 1974; and Norman, Rumelhart, and the LNR Research Group, *Explorations in Cognition*, 1975.] The Long-Term Memory of modern psychology is essentially a storage mechanism, although one which apparently utilizes some fairly sophisticated accessing strategies and can process requests in a sort of "background mode" while other processing is going on. [See, for instance, Norman and Bobrow, "On the Role of Active Memory Processes in Perception and Cognition".] These theories assign only a minor role to Sensory Memory; it is basically considered to be a storage buffer interfacing the perceptual

organs with the brain. Much mental processing (whether of incoming, previously stored, or internally generated information) is mediated through the Short-Term Memory, and the discussion that follows will focus on this level. Where an Intermediate-Term Memory is posited, it is seen as a storage facility that deals with transitory material of recent minutes or hours in contrast with the more permanent storage of Long-Term Memory.

The simplest model is probably the one originally proposed by Broadbent. [*cf.* the extract from Broadbent, *Perception and Communication* in Norman, *Memory and Attention*, pp. 22-24 and Norman's comments thereon.] He viewed the nervous system as a single-channel structure employing a filter to select from incoming information that which seemed relevant to current processing. He believed that selection was based exclusively on the physiological characteristics of the sensory input. A Short-Term Memory would be required for the temporary retention of information that had already been obtained through the senses but which hadn't yet been passed by the filter. This theory was influential [Norman, *Memory and Attention*, p. 22.], but its claim of single-channel intake and the requirement that switching of attention be based entirely on low-level physiological factors were effectively disputed [Norman, *Memory and Attention*, pp. 24-26.]. Still, the model did provide a valuable starting point for subsequent work.

In this context, debate continues about the number of different kinds of memory and the characteristics of each type. The relative permanence of their contents is not the only factor distinguishing the different types of memory: the kind of information that is stored, the mode in which it is represented, and the mechanisms for its storage and retrieval are also considered. Researchers in several fields are studying a variety of related physiological and psychological mechanisms, including those involved in perceptual, linguistic, episodic, and pictorial memory. As research proceeds, the recognition is growing that sensory mechanisms are affected by the content and requirements of the brain's processing. This contrasts sharply with the age-old view of perception as a passive process which simply receives incoming data from the world, and this new perspective has widespread implications for cognitive theories. [See, for example, MacKay, "The Epistemological Problem for Automata"; Gregory, *Eye and Brain*, pp. 11-12 and Chapter 12; Ornstein, *The Psychology of Consciousness*, Chapter 2, particularly pp. 35-36; and Norman, *Memory and Attention*, Chapters 2 and 3.]

The traditional psychological research into memory focused primarily on recall, ignoring more complex kinds of mental processing. Typical experiments confronted subjects with lists of symbols to be memorized, then tested the ability of the subjects to recall these symbols. The factors that were varied in these experiments included characteristics of the data, conditions under which memorization took place, and

constraints on the recall process. The model underlying the traditional experimentation viewed Short-Term Memory as the psycho-physiological structure in which most mental information processing took place. Short-Term Memory was also seen as the facility mediating between Sensory Memory and Long-Term Memory, organizing and selecting the information issuing from the former that would be retained by the latter.

These theories presented a picture of relatively simple processes in Sensory and Long-Term Memory that serve mostly to support the main cognitive processing occurring in Short-Term Memory. An example of this supporting role is the way associative recall and "background" handling of access requests contribute to the main processing, as, for instance, when a name is recalled or a problem is solved long after one has given up on the effort. This scheme accords with an intuitive conception of "consciousness" or "awareness" or "thinking" which envisions a stream of perceptions, thoughts, and memories flowing through the mind. Although the possibility that much processing may take place beneath this level of awareness must be recognized, the orientation of the naive view is similar to that of traditional psychology.

B. Limitations on the Capacity of Immediate Attention

The results of psychological research that are important for the present discussion are those which reveal strong, inherent limitations on the capacity of a person's Short-Term Memory. The following statement is representative of a major theme pursued in the contemporary literature on the topic of memory and attention: "how much do we retain of events that have just previously occurred? . . . When we try to determine exactly how much is retained of an immediate memory, we find very little. . . . In fact, the results of a large number of carefully controlled experiments indicate that humans can usually recall only a very limited number of items which have just been presented to them -- from as few as four, to, perhaps, ten items. . . . The contents of our immediate recall of unrelated material is commonly called the memory span, but this is a misleading phrase; it should be called the memory limitation." [Norman, *Memory and Attention*, p. 85.]

Contemporary cognitive psychologists address a wider range of phenomena than those involved in the traditional recall experiments, emphasizing the *processing* of information more than its *storage*. Still, according to the contemporary view, human information processing occurs primarily in Short-Term Memory. The discussion in later parts of this report and the current interest in information processing in the field of Cognitive Psychology call for a conceptual approach which goes beyond the traditional concern with immediate recall. Recall refers to the ability to absorb, retain, and reproduce information within short periods of time. Sometimes one needs to

remember an item during only a brief interval, as when dialing a telephone number obtained from a directory. The scope of Short-Term Memory extends beyond its use in this sort of immediate recall: the information it holds provides the material for the mind's central cognitive processes.

This essential function of Short-Term Memory is obscured by the emphasis on recall tasks in traditional experimentation. Therefore, the term **Immediate Attention** is proposed here to encompass both the storage function of Short-Term Memory and the processes of organization, selection, and manipulation which are performed on its contents. [After this term was chosen, it was encouraging to find one psychologist giving Sensory Memory the name "Preattentive Memory" {Kroll, "Visual Short-Term Memory", pp. 166-172.}.] Recall forms a core on which the other activities of Immediate Attention depend. The recall function of Short-Term Memory (in conjunction with the other levels of memory) provides the raw material for other processes and serves as a temporary storage for their partial results. Consequently, any quantitative limitations on the storage capacity of Short-Term Memory will of necessity restrict more general processing of information in the human mind/brain.

An instructive analogy can be made between human information processing and the operation of an electronic computer. The computer equivalent of Immediate Attention would be the set of registers that contain either the operands of computational operations or partial results that are expected to be useful within a short period of time. The large amounts of data stored away in the bulk memory devices of the computer would represent Long-Term Memory. The buffers interfacing the input devices to the central processing unit would correspond to Sensory Memory. The computational analogy is actually more amenable to a four-way division, as a computer's main memory is considerably more accessible and flexible to manipulate than the devices such as disks or drums which are also used in computation. (Information on tapes, cards, and in other "peripheral" media would then be like the information we store in books, recordings, and the like.) The essential point here is that there is a certain set of entities which are held in Immediate Attention and that Immediate Attention is where most information processing takes place. [In a similar context, Newell suggests that "What Waugh and Norman call primary memory {Waugh and Norman, "Primary Memory," 1965.} would be called a scratchpad memory or working memory." {Newell, "Production Systems", p. 466.}] These entities have either just been obtained from an input device (analogous to perception), or have just been extracted from the main memory (analogous to recall), or are the results of the manipulation of other elements in the Immediate Attention set.

There are two considerations in computer architecture that almost invariably lead to having this small set of special registers. In the first place, the registers are constructed out of memory devices with extremely fast access speeds. Because of factors such as cost, power consumption, thermal and electromagnetic characteristics, and instability (depending on the actual technology at issue), these devices are not appropriate for use in great quantity within the computer. Secondly, the hardware of the machine is greatly simplified by the requirement that data be extracted from the main memory into some register before it can be operated on. Readers who are familiar with the architecture of particular computers should note in this regard that these special registers include not only the accumulators and index registers accessible to the machine-language programmer, but also "hidden" registers such as those containing the contents of an addressed memory word.

All processing utilizes these registers, including transfers of information from one location in the main memory to another, transfers of information from the main memory to a peripheral device (output), transfers of information from peripheral devices to the main memory (input), and the arithmetical and logical operations of the machine. The program itself is executed under the control of several of these registers, which contain items such as status flags and the address of the next instruction to be executed. A significant component of the time required for a computation is the time it takes to obtain data from its storage location, in many cases one of these registers. Clearly, the efficiency of the machine is greatly improved if those relatively few elements which are involved in most operations have exceptionally fast "access times". The increased cost for the more expensive technology needed to achieve this is mitigated by the relatively small number of registers involved. In turn, the relatively great speed of the special registers makes feasible the standard architecture, with its requirement that all data move through one of a small number of elements.

This analogy can help illuminate the present discussion concerning quantitative limitations on the Immediate Attention set. A typical computer might have sixteen accumulators, a program counter, a program state register, a memory address register, and a memory access register -- twenty in all. In such a computer, no means exist by which fifty numerical quantities, each requiring one register for storage, could be simultaneously maintained in Immediate Attention. As the computation progressed, some pieces of data would have to be shifted back to the main memory for a time while others were manipulated. However, it is no more taxing to store a small integer in one of these registers than it is to store a huge number and no more difficult to store a five-letter word than a single character, within certain information-theoretic limitations on the maximum amount of information that can be stored in a register of

a given size. Although this may seem trivial and somewhat obvious, it has far-reaching consequences. George Miller established a limitation on human Immediate Attention capacity in his now classic article "The Magical Number Seven, Plus or Minus Two". He discovered that by combining several symbols into just one higher order symbol (for instance, three binary digits could be represented as a single octal digit), the amount of information held in Immediate Attention as a whole could be increased, but that in all cases, the number of individual entities stored was limited to less than ten or so.

Research since Miller's has developed this theme in more detail. There is some indication that the limit is more restrictive than at first realized, perhaps allowing just two compound entities along with several simple ones to be maintained together in Immediate Attention. [See, *e.g.*, the discussion in Simon, "The Psychology of Thinking," in his *Sciences of the Artificial*, especially pp. 39-40.] The exact number is not nearly as important as the fact that there does seem to be a very low limit on the number of entities one can simultaneously hold in Immediate Attention. What makes humans able to process huge amounts of complex information as they move about the world doing whatever they do is that pieces of information are combined into higher structures, a process that could be called folding. Consider a game: a sequence of digits is read and one must repeat them in reverse order. Virtually no-one could successfully repeat the sequence 92587274204377294. It contains seventeen digits; by the time the last one is heard the earlier digits would have been forced out of Immediate Attention by the ones which had arrived more recently. However, for seventeen digits that have some semantic interrelation, backwards repetition can be a trivial task. The sequences 12345678987654321 and 11112222333344445 are fairly easy to recall. In addition to its contribution to short-term recall, useful semantic structure facilitates the efficient storage and retrieval of information in Long-Term Memory.

This phenomenon can be informally explained by noting that all the digits can be grouped into just a few structures, representing something like "from 1 to 9 and back down again". These compound structures would take up only a few of the available "slots" in the Immediate Attention set, and after more processing occurred might require just one slot. [There are all sorts of theories about which information is discarded as new information comes in and how the decision is made. See, for instance, Gilmartin, Newell, and Simon, "A Program Modeling Short-Term Memory Under Strategy Control," pp. 15-30; and Newell, "Production Systems".] A sentence may contain thirty sound-units (phonemes) but the meaning can still be understood through folding. Several phonemes are folded into a recognized word, several words are synthesized into a phrase, and the phrase takes its place in the sentence as a whole. By way of a naive example, say that the phonemes of the phrase "the squirrel over there" require six slots, one for each syllable. As comprehension

proceeds, words are quickly extracted from the sequence of sounds, reducing the storage requirement to four slots. Then, with proper understanding of the phrase in conjunction with the perception of a squirrel over there, the information in the four slots are compacted into a structure that can be stored in one slot alone. This folded structure represents the actual referent of the phrase, the squirrel that is seen over there. [Experiments in linguistic memory show that often the exact words of a sentence are forgotten while the sentence's meaning is accurately recalled many days later. {*cf.* Wanner, *On Remembering, Forgetting, and Understanding Sentences.*} This is further evidence demonstrating that information is folded into compound conceptual structures.]

Thus, the quantitative limitation on Immediate Attention strongly conditions the way the human brain/mind processes information. Processing strategies must make efficient use of the small number of available slots. There must be interaction between Immediate Attention and Long-Term Memory, even for purposes of relatively short-term recall on the order of seconds or minutes after initial storage. The theme of quantitative resource limitations in human information processing will surface repeatedly throughout this report. Specifically, these limitations are significant for problems involved in understanding the behavior of complex systems because unless monitoring tools are carefully designed in keeping with these facts about human information processing, they tend to overwhelm their users with too much information at too low a level. For the user to comprehend the complex activity he is observing, it is crucial that he receive relevant information in a form that allows much to be held in Immediate Attention, despite the quantitative limitations on that facility.

Furthermore, although the process of folding can effectively reduce the number of slots taken up by a particular body of information, those slots remain unavailable for other uses while the folding takes place. The higher the conceptual level of information presented to a person, the less storage it initially requires in Immediate Attention and the more quickly the ultimate compound representations can be achieved. Psychologists have noted that sets of low-level perceptions may be suddenly organized into higher-level conceptual structures. It is postulated here that for any particular conceptual level, a certain threshold body of information must be immediately and simultaneously present to Immediate Attention for this organizational process to occur. Sequential observations are not as useful as when the same observations can be held simultaneously "in the mind's eye", allowing more effective apperception of interrelationships and patterns. [This will be discussed further in Section C of Chapter 7 below.] One theme of the present work is that monitoring systems gain power to the extent that they pre-fold information for the user, so that more pieces of the whole picture can be maintained in the user's Immediate Attention at once. The material of this chapter will serve to ground the discussion of this and and related themes in what follows.

Chapter 3

Computer Programs

This chapter outlines some fundamental aspects of computer programs and discusses their significance in relation to the characteristics of human information processing considered in the previous chapter. Despite the fact that much of this material will be familiar to most readers, several factors make it important to deal with it here. First, there are many ways of looking at programs and programming, and this chapter clarifies the viewpoint from which this work was performed. Then, some standard features of programs can be understood in a new light when considered in the context of human information processing. Finally, the analysis in much of the rest of this report depends on the particular conception of computational behavior that is explored in Section C of this chapter.

A. The Nature of Computer Programs

Electronic computers are instruments used for the manipulation of numerical and symbolic information. They are capable of recording on their various memory devices large amounts of coded information and can reliably perform simple operations on this information at great speeds. It is the speed, memory capacity, and reliability that make computers suitable for the uses to which they have been put: the operations performed by a computer's central processor unit (CPU) are remarkably simple. Typically, a single operation involves activity on the order of adding two numbers, transferring a symbol from one memory location in the machine to another, or comparing the magnitude of two numbers.

At the level corresponding to the basic operations which the computer is engineered to perform, a computer **program** is essentially an ordered list of binary-coded instructions, each of which represents the execution of one particular **operation** (such as ADD) on a small number of specified **operands**. When a computer program is executed, these instructions are **decoded** by electronic circuits that determine which operation is represented by the instruction and what particular operands are specified. The computer circuitry decodes instructions and performs the indicated operations one after the other until a program instruction, hardware malfunction, or human

intervention causes the execution to terminate.

Computer programming is the act and art of devising an appropriate set of instructions (a program) to cause some desired computational behavior to occur when the program is executed on a computer. As the binary code of the computer is far too clumsy for a human programmer to work with, programming languages have been developed which facilitate the expression of programs. In the simplest of these -- assembly languages -- each instruction essentially corresponds to a single machine language (binary-coded) instruction, but is expressed in symbolic terms more like those of human languages. For example, the Digital Equipment Corporation's PDP-10 computer has instructions which are thirty-six bits long. The first nine bits of information indicate which operation is to be performed. The next four bits specify a particular register as one of the operands of the instruction. The last twenty-three bits indicate the location of the other operand in the computer's memory, expressed in terms of an address, an index register, and an indirect bit. The content of the index register is added to the specified address. If the indirect bit is off, then the contents of the location specified is used as the operand. If the bit is on, the value is interpreted as another twenty-three bit location specification, this process continuing until a location is reached in which the indirect bit is off.

Symbolic Instructions. An instruction to add the contents of register 5 to the contents of memory location 33241 and store the result back in register 5 would be expressed in PDP-10 machine language as:

```
010111000010100000000011011010100001
```

More convenient representations of binary numbers use octal (base-eight) or hexadecimal (base-sixteen) number systems. It is particularly easy to convert a binary number to octal representation, since each octal digit corresponds to exactly three of the binary digits. Although octal numbers are easier to read than binary, the above instruction expressed in octal representation is still unwieldy:

```
270240033241
```

In the MACRO-10 assembly language for the PDP-10 [DEC, *PDP-10 Reference Handbook*.], this instruction can be represented as:

```
ADD 5, 33241
```

Furthermore, the programmer can define symbols that represent addresses, so the above instruction can be simplified to something like:

```
ADD 5, NXTLOC
```

The task of composing and understanding instructions is far more manageable using the kind of symbolic forms shown in this last example than if one had to deal

with the cruder representations that were shown first. The mnemonic significance of symbolic representations facilitates both writing and reading such instructions. It is important to understand that although humans can deal with such symbolic instructions more easily than with binary coded ones, the representation is still that of a single instruction indicating one basic operation of the computer. The issue here is only one of form -- very simple programs, called assemblers, translate instructions in this human-oriented form to the binary representation which the computer's hardware is designed to interpret. The development of programming languages has been strongly influenced by the advantages gained when computer programs are made easier for people to design, write, and read. Some of the more important steps in the expansion of the expressive power of programming languages are described next.

Macros. Many programs are written in assembly languages. It became apparent early in the history of programming languages that it was unnecessarily restrictive to require that each human-oriented instruction refer to a single machine-level instruction. As long as a program was required to translate from one form to the other, that program might just as well be expanded to perform other manipulations. Macro instructions were one of the first such extensions to be developed. Macro facilities permit a name to be attached to a particular sequence of instructions. Then, every time that same sequence must be repeated in the program code, the name alone can be used. The assembler replaces the macro with the sequence of instructions the macro represents. [Rosen indicates that the term "macro instruction" was coined at least as early as 1956 in connection with AUTOCODER libraries for the IBM 702/705. {Rosen, *Programming Systems and Languages*, p. 15.}]

Macro expansion was extended to allow parameterization, wherein each repetition of the group of instructions could be modified according to symbol substitutions specified along with the name of the macro. The use of macros saves the programmer from having to retype sections of code that are used frequently. In addition to saving the programmer time and effort, macros help avoid typographical mistakes that might otherwise be made while copying a piece of code from one part of the program to another. Furthermore, since complex operations can be invoked in one line of assembler code, imposition of this extra level of organization improves the readability of complicated programs.

High-Level Programming Languages. As the trend toward making program instructions more symbolic continued, high-level programming languages were created. FORTRAN was one of the earliest to achieve success (the project began in 1954). Its designers wrote: "The goal of the FORTRAN project was to enable the programmer to specify a numerical procedure using a concise language like that of mathematics and

obtain automatically from this specification an efficient 704 program to carry out the procedure. It was expected that such a system would reduce the coding and debugging task to less than one-fifth of the job it had been." [Backus *et al.*, "The FORTRAN Automatic Coding System," p. 29.] The first example in the original report on the FORTRAN project was one which became the standard illustration of FORTRAN's advantages. The formula for one of the roots of a quadratic equation

$$ax^2 + bx + c$$

can be simply expressed in FORTRAN as

```
ROOT = (-(B/2.0)+SQRTF((B/2.0)**2-A*C))/A.
```

Subprograms. Another concept that was recognized quite early in the development of programming practices was that of **subprograms**, also called **functions** or **subroutines**. The idea behind this concept was to isolate some piece of the code which performed a separate, coherent computation and give that piece a name. Then, as is the case for macro instructions, whenever that code is needed in the program it could be referred to by name alone, perhaps along with a few parameters. Subprograms differ from macros in that they are "called" by special statements which cause control to be transferred to another part of the code. Then, after the instructions in the subprogram have been executed, control is returned to the next statement following the call. Thus, along with the advantages gained by using macros, subprograms provide the additional benefit that the machine level instructions into which they are compiled are not repeated at each place in the program where they are called. This gain in efficiency is particularly important for frequently computed functions, such as square roots, random number generators, and trigonometric functions which would otherwise constitute substantial portions of the machine code of some programs.

Other benefits accrue from the judicious use of subprograms, even in the case of computations that are not repeated during any one execution of the program. Errors from one logically distinct section of the code can cause problems in a following section. Breaking down the program into its constituent parts reduces such problems of error compounding by isolating the code and local storage of each subprogram from that of the others. Useful subprograms can be used again in other programs; in fact, most programming language environments include **libraries** of generally useful subprograms. These libraries save programmers the effort of duplicating each other's work and provide a better implementation of specific functions than the individual programmer would usually devise. Finally, since each subprogram can be compiled and tested separately, changes to one do not require recompilation and retesting of the other subprograms in the program.

B. The Significance of High-Level Programming Constructs

The development of programming languages traces out an historical and conceptual hierarchy, from binary-coded machine languages through symbolic assembly languages to high-level programming languages. There is also at each language level a structural hierarchy: programs are composed of macros or subprograms which are in turn composed of individual statements. [Macros are not restricted to assembly-level languages; SAIL, for instance, has a well-developed macro facility.] Figures 1 and 2 summarize these two hierarchies. In both the language and structural hierarchies, an instance of a construct at one level is composed of one or more instances of a lower level construct.

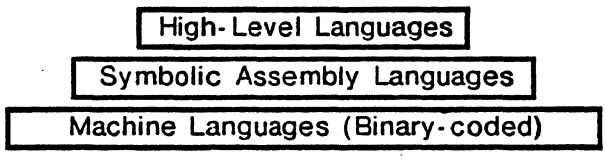


Figure 1. The Programming Language Hierarchy

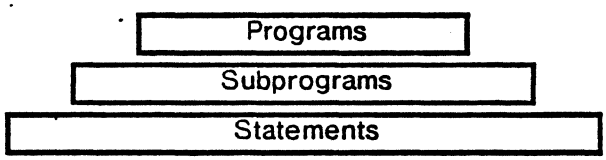


Figure 2. The Program Structure Hierarchy

Greater Conceptual Power at Higher Levels. Although usually each high-level construct is compose of *many* low-level constructs, this is not always the case: an individual assembly language statement often represents a single machine-level instruction, and sometimes a simple statement in a high-level language compiles into only one machine instruction. The salient dimension of this hierarchy is conceptual power, not strictly composition (although composition does contribute to conceptual power). A construct at one level communicates more in the same amount of time and space than do lower-level constructs. People find it easier to understand a symbolic assembly language statement than an octal machine instruction, even though they specify the same operation. By writing one ALGOL iterative statement, the same computation can be specified that would require a dozen or so assembly language statements.

The greater conceptual power of higher-level constructs requires some explanation. A first-order theory of comprehension might assume that an instance of a higher-level construct is simply decomposed into instances of constructs at the next

lower level, with the processing continuing until symbols at some basic level -- binary-coded machine instructions, in this case -- are obtained. Such decomposition is certainly *part* of comprehension processes, and compilation of a high-level language statement into several machine-language instructions is in fact essentially a decomposition process. [In this light, the term compilation seems somewhat of a misnomer, since the process is largely one of decomposition.] However, this first-order theory does not recognize limitations on processing resources that are an important factor in accounting for the greater conceptual power of higher-level constructs.

The quantitative limitations on human Immediate Attention that were discussed in Chapter 2, and the similar limits on the number of fast registers in a typical computer, bear on this issue. Consider the following ALGOL statement:

```
IF MAXVAL/3 > SQRT(SUM) THEN WRITE("ANSWER =", MAXVAL)
```

Structures representing most, if not all, of its details can be stored in human Immediate Attention together, because the statement includes only a small number of identifiers and operators. In contrast, the equivalent sequence of machine instructions would contain more information than could be represented at one time in Immediate Attention. In this case, temporary structures would have to be constructed and longer-term memory would have to be used, reducing the efficiency of the understanding process.

Although more *meaning* is communicated for a given amount of resources using high-level constructs than low-level ones, there is actually more *information* in a set of low-level statements than in a single high-level one meaning the same thing. [No widespread agreement has been reached in relevant fields on the relationship between information and meaning. One researcher made a proposal that accords nicely with the present sketch, suggesting that meaning and information are inversely related. {Crosson, "Information Theory and Phenomenology," especially p. 111 *ff.*} In another essay, the same author states that "Psychological evidence seems to confirm the hypothesis that the perceptual process is more correlated with meaning . . . then with information." {Crosson, "Memory, Models, and Meaning," p. 193.}] A major source of the conceptual power of higher-level constructs is that they filter out unnecessary information which would otherwise burden a person's limited mental resources. Given code to sum the numbers contained in a list, which register holds an iterative index and which contains the sum being calculated are details that are irrelevant to an understanding of the procedure. Writing or reading these distracting details uses up valuable mental resources to no advantage, where a higher-level statement can express the same meaning with less detail.

Facilitation of Task Stacking. The greater expressive power of high-level constructs is especially important for the programmer's ability to "stack" tasks, a skill

acquired with experience and one which is fundamental to the programming process. In this process, a current task is divided into subtasks. Only one of those subtasks can be pursued at any time; the others are "stacked" -- remembered -- for later processing. Then, the subtask that is being pursued may itself be divided into several sub-subtasks, and all but one of these will also have to be stacked for later processing. The notion of "stack" refers to the fact that the most recently stored activities are the ones resumed first, resulting in what is called a "depth-first" processing of the task divisions.

The higher up in the language and program hierarchies that one can terminate the division of subtasks into further sub-subtasks, the less is the burden on the programmer's ability to stack these tasks. More meaning can be represented with the maximum number of higher-level constructs that fit into Immediate Attention than can be with a full complement of low-level constructs. It would seem that one can stack effectively beyond the quantitative limits of Short-Term Memory, presumably by using Long-Term Memory. [This area of investigation is beyond the scope of the current work, but would be a good area for research.] Even so, recall processes would have to "reload" Short-Term Memory with previously stored intermediate states when they were needed in Immediate Attention processing.

Reduction of Programming Errors. Finally, the cost of programming errors is another significant factor in this context. Over the years, various attempts have been made to discover the parameters determining the frequency with which programmers make simple kinds of mistakes. Conventional wisdom in the field holds that the number of errors in a program is roughly proportional to the number of characters in the code. [The size of the program is often expressed in terms of lines of code as well. One author reports a common view that the number of errors is proportional to the number of labels {Schwartz, "An Overview of Bugs", p. 3.}, another way of approximating the size of a program.] The higher the conceptual level of the language used, the fewer the characters necessary to express a given meaning, so higher-level constructs help reduce the number of simple mistakes in the specification of a given computation. Furthermore, the higher the level of the language, the greater the cognitive significance of the elements of the code and so the easier it is to notice discrepancies. For instance, a mispelling of a symbolic identifier is more likely to be noticed then a mistyped number. High-level languages thus reduce the frequency of low-level mistakes in program coding and make the ones which *are* made easier to detect.

The designers of the first major high-level language recognized the factors discussed in this section which contribute to the value of such languages. They wrote: "The experience of the FORTRAN group in using the system has confirmed the original expectations concerning reduction of the task of problem preparation and the

efficiency of output programs." [Backus *et al.*, "The FORTRAN Automatic Coding System," p. 29.] Their report concluded:

> The statement of a program in FORTRAN language rather than in machine language or assembly program language is intended to result in a considerable reduction in the amount of thinking, bookkeeping, writing, and time required. In the problems mentioned . . . , the ratio of the number of output machine instructions to the number of input FORTRAN statements for each problem varied between about 4 and 20. (The number of machine instructions does not include any library subroutines and thus represents approximately the number which would need to be hand coded, since FORTRAN does not normally produce programs appreciably longer than corresponding hand-coded ones.) . . . The convenience of using FORTRAN language is necessarily more difficult to measure than its conciseness. However, the ratio of coding times, assembly program language vs FORTRAN language, gives some indication of the reduction in thinking and bookkeeping as well as in writing. This time reduction ratio appears to range also from about 4 to 20 although it is difficult to estimate accurately. [Backus *et al.*, "The FORTRAN Automatic Coding System," p. 45.]

C. Computational Behavior

Programs are inherently directed at the production of certain behavior on the part of the machine which executes the program's instructions. Whether a program is a good one or not depends primarily on whether it "works" -- that is, produces the behavior intended of it. A secondary consideration in evaluating a program is the efficiency with which it uses computational resources. The act of programming a computer and the act of determining what the behavior of the computer actually is during the execution of the program are naturally related, yet quite different. By way of convenience, programmers often speak of the behavior of the *program* when what they really mean is the behavior of the *computer* as it executes that program. This section discusses the notion **computational behavior**, the behavior of a computer as it executes some program.

Input/Output Pairs. As the purpose of most programs is to absorb information provided to the computer and manipulate it to produce new information, one standard way of defining the desired behavior of computer programs is through the specification of what are termed **input/output pairs**. The acceptable input expressions are specified, and the output which the program should produce is defined for each input. This definition can usually be generalized into a function which expresses the relationship between *classes* of input expressions and *classes* of output behavior.

To the extent that a program is viewed as a mechanism for accepting input data and producing "answers" -- whether the context is a simple numerical calculation or a highly complex Artificial Intelligence system -- "program behavior" can be characterized in terms of *overt actions* (printouts on some peripheral device, say). This is the stance which the program *user* takes when considering computational behavior. It is not, however, the viewpoint of the program *implementer* or *maintainer*. The

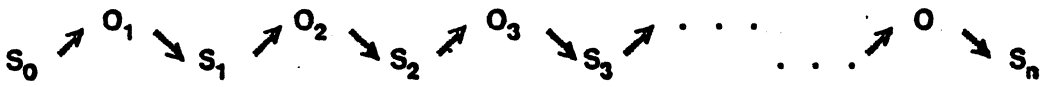
processes of programming and debugging require that behavior be defined not in terms of the gross output characteristics of the program, but in terms of the program's internal activity as it works its way through the indicated computations. [A less important reason for abandoning the use of input/output pairs in defining program behavior is that although the concept is often used in theoretical discussions, from a practical standpoint this form of specification is useful only in the simplest of cases.] The rest of this section defines a notion of computational behavior that is appropriate to the activities of programming and debugging.

Computational Sequences. Computer memories contain many tens or hundreds of thousands of words, each one representing a small unit of information -- one or two instructions, a number, a few characters, and so on. Each computer operation makes use of a few words of information and the result of the manipulation is stored in one or perhaps several locations (words) within the machine's memory. Thus, the execution of a single instruction results in only a *local* change to the contents of the machine's memory, leaving virtually all of memory unaffected. The notion of a data state is implicit in this way of talking about the contents of the machine's memory. At this basic level, a data state constists of the contents of the machine's entire memory at a time between instruction executions.

Computation is generally viewed as an essentially discrete sequence of steps, each one causing small changes to the data state of the machine as a whole. Note that a computer is a physical device and as such has no discrete states. This reveals yet another, lower level in the computational hierarchy. In the context of program behavior, state-transitions of the circuitry which occur during instruction execution are regarded as "below the grain" and ignored. A further distinction can be made along the same lines between states of the circuits and states of the individual solid state devices. A transistor is regarded as either on or off, depending on its output voltage. Of course, for this voltage to get from the "on" value to the "off" value or *vice versa* it must go through intermediate values. These intermediate values are not meaningful at the level of circuit states, just as the circuit states during execution of an instruction are not meaningful at the level of the machine's data state.

A computer program is a sequence of symbols which expresses the operation, operands, and destinations constituting each of the discrete computational steps. A computer program can be viewed as representing a sequence of transformations to be performed on the data state of the machine, and its behavior can be defined in terms of that sequence. There is a parallel perspective from which computational behavior can be defined in terms of the sequence of states that result from each transformation. So, the question "what computation is being performed?" can be answered either in

terms of the sequence of operations executed or in terms of the sequence of states that result. In either case, *the definition of computational behavior appeals to a fundamental notion of sequential, discrete change.* The operation- and data-oriented viewpoints are, of course, fundamentally interdependent: each operation causes the transition from one data state to another. Although there are two separate sequences, the computation could be in principle completely described in terms of either one. Figure 3 illustrates the two parallel sequences and their relationship.



Data States are denoted by S_i, Operations by O_n

Figure 3. The Parallel Sequences of States and Operations

More formally stated, each instruction is a function that takes a memory state of the machine into another memory state of the machine. The function composition rule for these functions is the standard one: $g \circ f(x) = g(f(x)) = h(x)$. Thus, by considering each instruction as a function which can be combined with the following instruction-functions, a program can itself be interpreted as a (composite) function taking the starting state of the machine into another state, which can be called the termination state. [See Knuth, *The Art of Computer Programming, Volume 1,* pp. 1-9 and Dijkstra, *A Discipline of Programming,* chapters 0-4 for more thorough discussions of the formal definition of "program".] In this respect, the skill of computer programming involves knowing how to put operations together so that the transformation thereby specified will produce a desired termination state from any appropriate initial state (although it is generally not useful for the programmer to take this formalistic perspective).

Conditional Operations. The model of computation sketched so far is a strictly linear one: the machine performs the first operation, then the second, continuing until the last. Programs are ordered lists of instructions that specify which operations are to be performed. The descriptions presented to this point imply that when a program of N instructions is executed, N operations will be performed. These descriptions have omitted a crucial aspect of computer programming which must be taken into account in defining computational behavior. Program code and data states interact to produce computational sequences not explicitly designated in the code. There is in any programming language, at whatever hierarchical level, a way of specifying **conditional operations.** The concept of conditional operations is so basic to computer programming that virtually every program contains such statements. If a program contains even *one* conditional statement, the program no longer specifies a sequence of operations for the

computer to perform, but rather a *set* of sequences.

In the simplest cases, conditionals are composed of a test and an operation; the operation is either performed or ignored, depending on the results of the test. The tests are functions of the data state, but usually of only a very small part -- typically they depend on the contents of just one or two memory locations. Common tests include whether or not a word is equal to zero and which of two numbers is larger. More generally, a conditional statement may specify a set of pairs, each containing a test and an action; for tests which "succeed", the associated action is performed. For a particular construct in a some language, there may be various restrictions on the tests, operations, and sets of their pairings. For instance, it may be required that the tests be mutually exclusive and that at least one will succeed. Such is the case for the FORTRAN IF statement. It contains an expression and three statement numbers. The expression is evaluated and the action taken depends on whether the resulting numerical quantity is negative, positive or equal to zero. Other languages have constructs that allow arbitrarily large sets of pairs (case statements). In this kind of conditional, the tests are considered sequentially, and the operation associated with the first one to succeed is performed.

Branching and Looping. Conditional statements are often used for **branching**: when the test is reached during execution of the program, the program continues at one of several specified points, depending on the results of the test. For example, a numerical quantity may need to be treated differently if it is negative than if it is positive. Branching is commonly employed to cause **looping**, whereby the program continues execution at a statement it had already executed once before. Obviously, for a program to terminate, any loops entered must also be exited. It is a rare program that contains no loops at all, although the power of high-level language statements sometimes disguises them. Typical kinds of loops are iterations, in which part of the program is repeated for each of a specified set of values, and loops that are repeated indefinitely until some test condition becomes true.

The length of the sequence of operations performed during the execution of a program that has no branches is exactly equal to the number of instructions the program represents. If the program contains branches, but no loops, then the length of the sequence may be *less* than the number of instructions represented. If the program contains a loop, than the sequence length is potentially greater than the number of instructions represented by the program. If the exit condition of the loop depends only on information contained in the program itself, then the potential length of the sequence has a clearly determinable bound. This would be the case for a numerical iteration that specifies in the code the starting value, the termination value, and the

increment to be added to the last value after each loop. However, if the loop termination condition depends on some feature of the input, then bounds on the sequence length can no longer be established simply by inspection of the program. In fact, it is in general impossible to prove that the program will terminate for all valid inputs, and even in those restricted cases where it is theoretically possible, the task tends to be impractically difficult.

As these generalizing factors are added to the basic concept of a computer program as an ordered list of instructions, computational behavior becomes more dependent on the particular input supplied for a given execution of the program and less determined by the program code alone. For programs of even moderate complexity, it may be not be possible in practice to analytically characterize their potential computational behavior over the entire range of valid inputs.. Thus, as program complexity increases, determining computational behavior becomes more of an empirical enterprise than a theoretical and analytical one. As will be shown in the next chapter, the observation of computational behavior constitutes a major portion of the activity involved in debugging a program.

Higher Level Notions of Computational Behavior. Computational behavior has been defined here in terms of progressions of machine-level operations and data states. Statements in higher-level languages are ultimately decomposed into machine-level instructions. Thus, a simple operation in *APL* -- the addition of two matrices, say -- is represented by a single character between the two operands, but its interpretation may produce tens of machine-level instructions which in turn, through iteration, cause hundreds or thousands of machine operations. Likewise, higher-level data structures consist of individual words of the machine's memory, or in some cases parts of those words. For example, a character string will utilize several consecutive memory words, while the LISP CONS cell is implemented on some computers as a single word divided into two halves. A computation can be described not only in terms of the fundamental operations wired into the computer's hardware, but also as a sequence of operations which are basic to some higher-level programming language. By the same token, computation can be defined as a sequence of data states defined not at the level of memory words, but in terms of the constructs in the higher-level language.

Therefore, higher-level notions of computational behavior are available, corresponding to higher levels in the programming language hierarchy. The underlying idea of computational behavior as a sequence of operations or of data states remains unchanged regardless of the level addressed. *The essential determinant in a definition of computational behavior for some language is the particular concept of a single discrete change in the state of the system that is inherent in the design of that*

language. The invocation of an assembly-language macro is often conceptually a single step in its program, but it results in the performance of many machine-level instructions. A single change to a data structure in some high-level language may involve separate changes to many individual machine-level words. [Two clarifying points must be made here. First, it is of course possible to design hardware (or microcode) that implements directly the basic operations of some high-level language. Several efforts aimed at producing a "LISP machine" are currently underway, most notably at the MIT Artificial Intelligence Laboratory. {See, *e.g.*, The Lisp Machine Group, *Lisp Machine Progress Report.*} Another such effort is reported in Bashkow, Sasson, and Kronfeld, "System Design of a FORTRAN Machine". Second, the concept of computational behavior defined here is oriented towards the traditional "von Neumann machine"; for other machine architectures and programming languages, perhaps ones still to be invented, other approaches to the definition of computational behavior may be more appropriate.] Computational behavior is thus a hierarchically relative notion. This has important consequences for debugging activity and the nature of complex computational systems, the subject of the next two chapters.

Chapter 4

Debugging Computer Programs

It is a fact of life for the computer programmer that when programs are first written they require substantial alteration before they will do anything meaningful, and that even then, the actual behavior of a program usually approaches the intended behavior only asymptotically. The term **debugging** covers almost everything that the programmer does between initially writing the program and declaring it "finished". This presumes that the program is written first in its entirety and then debugged; certain more interactive environments, for example that of INTERLISP, facilitate and encourage the incremental development of programs with the programmer filling in code as required. In such an environment, the acts of programming and debugging are more strongly intertwined. Furthermore, even *after* the program is declared "finished", much of the programmer's effort is devoted to debugging, as users of the program expose problems. This chapter discusses the nature of program errors, the tools traditionally provided for their detection and correction, and the nature of activity aimed at problem detection and correction.

A. Detecting Programming Errors

When a programmer writes a program, he has in mind an activity he desires the computer to perform. As noted at the beginning of the last section of the previous chapter, this activity is conceptualized at the level of the program's manipulations of its internal data structures, not just at the level of overt output behavior. The programmer conceives of a method by which the desired activity can be performed and expresses that method in terms of a programming language. Next, he communicates that expressed method to the computer through input devices such as a card-reader or teletype. Finally, he issues instructions to the computer which result in the execution of the program. In these respects, computer programming is an act of communication, and it breaks down in ways typical of all communicative acts. Seen from this perspective, five major types of programming errors can be isolated, distinguished by the aspect of the communication process from which they arise: lexical, syntactic, semantic, teleological, and conceptual.

The simplest programming errors arise in the physical act of transmitting characters representing the program to the computer. Misspellings, typographical mistakes, and physical errors in the communications channel are the kinds of simple errors which will here be called **lexical errors.** This term is derived from the fact that such errors interfere with the computer's recognition of individual words. More specifically, it refers to the part of a compiler (the program that translates a high-level language program into machine code) that is often called the **lexical analyzer** or **scanner,** as it is this part which fails to perform correctly when it encounters a character-level mistake.

Lexical errors are perhaps the most frequent, the easiest to detect by man or machine, and the easiest to fix of all the types of errors. In general, they are detected by the same mechanisms used to detect errors of the next class -- syntactic errors. In most programming environments the language interpretation process is *literal*: no account is taken of what the programmer is *likely* to mean by a particular string of characters, so lexical errors (misspelling a key word, for instance) almost invariably cause syntactic errors. The signalling of syntactic error conditions which are actually due to lexical mistakes is a constant irritation which is accepted and overlooked by most programmers. [A major exception to language interpreter literalness is provided by the DWIM (Do What I Mean) facility in INTERLISP. {See Teitelman, "Toward a Programming Laboratory" and the INTERLISP manual, section 17.} DWIM is an interface between the user's type-in and the language interpreter which is invoked when certain kinds of lexical errors are detected. Based on built-in knowledge about INTERLISP syntax, INTERLISP semantics, the layout of the keyboard, and the kinds of mistakes people tend to make, DWIM attempts to correct the errors it detects. For instance, if the user misspells a function name or types a number when the shift-character of that number was intended (*e.g.*, an eight instead of a left parenthesis on some terminals), the system can often correct the error.]

Each programming language incorporates rules governing the allowable forms of program statements. Violation of these rules are commonly called **syntactic errors.** When a program includes this kind of error, it is easy to detect *that* an error has occurred: the computer system will simply not be able to interpret the improperly formed statements. Determining that the problem was a *syntactic* error, and more precisely, *where* and *what* the error is, may be difficult. Procedures for notifying the programmer of syntactic errors are normally built into the compiler or interpreter. Most compilers attempt to continue the processing as far as possible after each such error, so that many syntactic errors can be caught on one pass. Frequently the compiler is only able to point out the general neighborhood of the error in the code. It also sometimes happens that a lexical or syntactic error can result in an expression that is coincidentally correct (although with an unintended meaning), so that a small percentage of such errors will not be caught by the detection machinery. In these odd

cases, errors of a higher order will result. Ferreting out such "promoted" errors is often a particularly tricky undertaking for the programmer.

Once the program is sufficiently correct lexically and syntactically, the program can be executed. Errors that often arise at this stage are those in which the *meanings* of the syntactic forms are unclear, contradictory, or invalid. Attempting to divide by zero, to multiply a number by a string, or to call a function which has no definition are typical **semantic errors.** Such errors are called **run-time errors** in many programming environments and are detected by the programming language's run-time support system in cooperation with the hardware of the host machine (by way of flags, interrupts, and the like). In a few languages the attempt is made to detect semantic errors at the syntax level, through "strong compile-time type checking of both data types and program interfaces" [Mitchell, Maybury, and Sweet, *Mesa Language Manual*, from the Abstract. Mesa, developed at Xerox Palo Alto Research Center, "is *strongly type-checked*; i.e., every variable used in a *legal* Mesa program must be declared to have some type; every constant has a type; and every expression has a type derived from its components and context. All types are known at compile-time, and the compiler enforces the correct use of variables by type. The information included in a variable's type can include value classification . . . structure . . . symbolic representation for constants . . . boundary indications . . . storage occupancy . . . [and] access method . . . ." {p. 4.}] Even though they may be detected by the compiler, the *sources* of errors violating type-check constraints nevertheless lie at the semantic level.

In contrast to the compile-time flexibility exhibited by most compilers with respect to syntactic errors, it is rarely possible for computation to proceed past a semantic error. The reliability of this "feature" releases the programmer from worries about *whether* his program has such undetected mistakes in it. As long as he has the resources to execute his incorrect programs as many times as is necessary to eliminate them, the programmer can be fairly confident that his semantic errors will be discovered by the computer. Some interactive languages (*e.g.*, INTERLISP) provide mechanisms by which the programmer can correct or bypass semantic errors to continue the computation. However, in most environments the programmer is left on his own to pursue the elimination of these errors once they have caused the execution to be terminated.

When the low order errors (lexical and syntactic) and the middle order errors (semantic) are weeded out, the more interesting and more challenging high-order errors rise to the fore. At this stage, it is often the case that the program does *something*, but not what was intended. The term **teleological errors** will be used here to refer to problems in the domain of the programmer's precise specification of the computational scheme or algorithm. For instance, the programmer may want to reverse the order of the elements in an array in place, but codes this procedure in such a way that new

elements are put into a location before the old ones are saved, instead of after. Here, the basic scheme is valid, but the implementation is faulty. Sometimes the source of the problem is in the method or approach devised by the programmer to achieve the desired computational activity. Errors of this last type will be called **conceptual errors.** High-order errors may be quite subtle and frequently are detected by users of the "finished" program rather than by the programmer. Figure 4 summarizes the five different types of errors and what or who normally detects them.

Class	Types	Detected By
low-order	lexical and syntactic	compiler
middle-order	semantic	run-time system
high-order	teleological and conceptual	programmer or user

Figure 4. Types of Programming Errors and their Detection

Program errors are not eliminated in strict hierarchical order. In most languages, low-order errors must be essentially eliminated before the program can be executed. (Because lexical errors are usually "promoted" to syntactic ones, errors of these two classes are generally handled at the same time.) In contrast, it is possible for middle-order errors to exist in a running program. One way for this to happen is for the actual input to lead to a branch in the program which bypasses the defective code. In other cases, the code may be defective only under certain conditions. The latter case could occur, for example, if some numbers are added together and used as the divisor in another step of the program. The programmer may have forgotten to insert a test to check that the numbers don't happen to sum to zero, and the program may be used for years without that particular problem being encountered. Thus, middle-order errors do not always interfere with the detection of high-order errors. The latter may be discovered in semantically correct sections of the program before sections still containing middle-order errors are exercised.

The upshot of all this is that each class of errors -- low-, middle-, and high-order -- is detected in a different way and presents a different sort of obstacle to correct program performance. Whereas low- and middle-order errors are for the most part detected by the computer system, the ordinary programming environment does little to detect high-order errors. Consequently, these must be discovered by human analysis of the program's behavior. [Work in "intelligent" programmer's assistants and automatic programming systems is just beginning to address this situation. See, for instance, Winograd, "Breaking the Complexity Barrier (Again)" and *Five Lectures on Artificial Intelligence*, Lecture 5; Hewitt and Smith, "Towards a Programming Apprentice"; and Rich and Shrobe, *Initial Report on a LISP Programmer's*

Apprentice.] The search for, and investigation of, high-order errors constitutes the bulk of the activity referred to as "debugging". More specifically, most debugging activity is directed at the teleological level; conceptual errors tend to be uncovered incidentally in the process of looking for teleological ones. Thus, and this will be discussed further below, most tools and techniques for debugging can only *facilitate*, rather than *perform*, the detection process, by providing the programmer with information to be used in the search for high-level errors and the attempt to understand their cause. In addition, although middle-order errors are effectively detected by the computer system, determining their cause is more difficult, and the available debugging tools will sometimes be useful for that task too.

B. Traditional Debugging Tools and Techniques

This section considers briefly the nature of traditional debugging tools and techniques. Other authors have presented good historical studies in this domain, and little would be accomplished by repeating those efforts here. [See in particular Satterthwaite, *Source Language Debugging Tools*, 1975, Chapter 2; Gaines, *The Debugging of Computer Programs*, 1969, throughout; and Evans and Darley, "On-Line Debugging Techniques," 1966.] Accordingly, the exploration in this section is thematic, rather than historical. The next section will discuss some recent developments in debugging techniques, especially those that relate to the main themes of the present work.

Traditional debugging tools have two principle foci, in accordance with the fundamentally dual aspect of program behavior, as discussed in Chapter 3. The two basic questions that one can ask about an ongoing computation are "What operation is being performed now?" and "What is the state of memory now?" Most forms of traditional debugging techniques provide straightforward methods for obtaining answers to these questions. The most primitive way of asking the state-oriented question is by means of memory dumps -- printouts which show the contents of all or some of the memory locations. These are often difficult to use because vast amounts of information are available at each point in computational time; consequently, refinements often address the problems of selective dumping.

The display console of most traditional computers includes a set of lights which indicate the instruction currently being executed. Whereas dumps present too much information at once, the procedural equivalent -- watching the console instruction display -- presents a very small amount of information that changes too rapidly over time to be of any use in program debugging [although in the earlier days of computers, machines were sufficiently slow that such information could in fact be helpful]. To obtain information about the instructions being executed the programmer must use special trace facilities.

For these too, it is usually best to be selective, tracing only a few (key) statements in the program code. Interactive debugging facilities are based on the idea of **breakpoints**; the program is halted at a specified point, whereupon the user gains control and can investigate the state of the program with special debugging commands.

As Gaines observed:

> There are three fundamental notions in debugging on which most of the tools and techniques used in debugging are based. These are dumps, traces, and breakpoints. A dump is a record of a large amount of information about the state of a computation at a given point. A trace is a record of the statements in the program executed in some interval during the computation, together with the result of each statement. Tracing gives information about the flow of control in this interval. A breakpoint is an interruption in the computation to transfer control to a debugging aid, which is specified independently from the program. The usual situation in which a breakpoint is used is that the program being debugged is operating under the control of a debugging system in which the programmer can specify that the program should be interrupted at some given point during the computation and describe the action to be taken when this point, called the breakpoint, is reached. [Gaines, *The Debugging of Computer Programs*, p. 223.]

Satterthwaite agreed, stating:

> Traditional debugging tools have been designed primarily to present information intended to aid detection and correction of mistakes in the logic of a program. Nearly all such tools, proposed or implemented, provide a more or less elaborate dump or trace under the control of some scheme of breakpoints. A *dump* is a partial or complete report of the state of a computation at a given instant, while a *trace* is a record, often largely implicit, of the sequence of state transitions occurring as a program is executed. *Breakpoints* are interruptions of the ordinary course of the computation to allow the initiation or modification of debugging activity. Some *debugging command language*, possibly quite trivial, must also be provided for specifying both the set of breakpoints and the action to be taken at each breakpoint. [Satterthwaite, *Source Language Debugging Tools*, p. 17.]

Some implementation of each of these three main themes in debugging techniques -- **dumps, traces, and breakpoints** -- has been developed in virtually every programming environment since the first programs were written. [See the historical surveys mentioned above for details.] The general approach to debugging that the progammer develops through the use of these tools can be called the **break-and-examine paradigm**. The program is stopped and its state examined, or a picture of certain parts of the state is recorded at specified points. There have been many variations on these themes, most of them simply improving the presentation of information or the mechanisms by which the user could specify what information he wanted to see.

Still, some important extensions and additions to these basic ideas have arisen. In many systems breakpoints are generalized to **conditional breakpoints**, where the break occurs only if a specified condition is met. (With respect to this generalization, breakpoints are analogous to jump instructions in program code, both in actual implementation and in that absolute and conditional varieties are available.) Another direction the generalization of this idea takes is toward automatic execution of tracing and dumping operations at the break, rather than returning control to the user. Most tracing facilities [*e.g.*, INTERLISP's TRACE function and various FORTRAN trace facilities] are

implemented either in effect or in actuality through the same mechanisms as are breakpoints, and differ from these only in that they give control to some indicated debugging routine instead of to the user.

The use of assertion statments (redundancy checks) is another standard debugging technique. They have been in use since in the earliest days of computing. [Satterthwaite points out {Satterthwaite, *Source Language Debugging Tools*, p. 20.} that assertions are mentioned by Goldstine and von Neumann as early as 1947 {in Goldstine and von Neumann, "Planning and Coding Problems for an Electronic Computing Instrument".}.] In theory, assertion statements might warrant separate status among the basic types of debugging techniques, but in actual debugging practice, assertions are used essentially as conditional breakpoints inserted directly into the program. [Their role in program verification, an area of computer science that will not be discussed here, is quite different.] Satterthwaite made much use of statement execution counts, sometimes called "flow counts". These are a form of trace in which information about execution frequency is presented along with each program statement. [As Satterthwaite mentions, these too were used by Goldstine and von Neumann.] Unlike ordinary traces, which present a linear flow of information as execution proceeds, execution counts are displayed along with a program listing after the execution has terminated.

One dimension for the classification of a debugging tool is whether it is oriented towards investigation of the data state or of the flow of control, as just discussed. Another useful dimension is whether or not the user is actively involved in the debugging process. There is a basic distinction between tools oriented towards batch processing and those designed for an interactive environment. The latter are often called dynamic, as in the acronym DDT, standing for "Dynamic Debugging Technique", and the former, by contrast, static. Actually, this is something of a misnomer, as "dynamic" tools involve freezing the computation to allow the user to investigate what is then a static state. Since "static" tools are inserted directly into or around program code, they are fixed for any given cycle of compilation and execution. In this sense, "static" is an appropriate term. However, in another sense, these tools can be considered "dynamic", since the program runs uninterrupted under the control of these tools, producing trace and dump information dynamically.

Better terms to replace "dynamic" and "static" would be interactive and non-interactive, as these more accurately reflect the basic distinction involved here. Note that the same tool can be used either interactively or non-interactively. An example is the INTERLISP BREAK package, which interrupts the program; arguments supplied when the package is invoked determine whether the user is to be given control or if instead, some operations are performed after which execution is resumed. Thus, although often closely correlated with the nature of the environment in which the debugging tool is

implemented, the distinction between interactive and non-interactive tools is essentially one of usage.

Traditional debugging tools help the user obtain information about the execution of his program. Given the dual nature of computation, the programmer naturally requires information about both the sequence of operations performed and the sequence of data states attained. Since he knows what he told the machine to do in his program, or can at least look at his program to find out, the programmer is not interested in a detailed record of every operation performed. Programs generally contain various kinds of branches and loops and therefore specify whole *families* of computational sequences. [See Chapter 3, Section C.] As a result, the programmer is often most interested in finding out *which* sequence was actually performed for some initial data state. Giving the user some picture of which of the many possible sequences of operations was actually performed is a major function of trace-oriented debugging tools.

In presenting information about the computational sequence, minor steps between decision points can be ignored. [*cf.* Gaines, *The Debugging of Computer Programs*, pp. 40-68 *passim* and Yarwood, pp. 11-13 and 23-24.] The recording of all subroutine calls, branches, and loops is sufficient to specify the complete sequence of operations that were actually performed during execution. Gaines mentions the intriguing concept of a "dynamic flowchart", wherein only the paths actually taken by the computation are shown. [Gaines, *The Debugging of Computer Programs*, p. 50. Incidentally, as Gaines points out {p. 48.}, flowcharts are yet another aid for understanding programs which were mentioned in Goldstine and von Neumann's article "Planning and Coding Problems for an Electronic Computing Instrument".] A dynamic flowchart would filter out unwanted information. It could be used to deal with self-modifying programs, which, because of their dynamically changing structure, cannot be represented by traditional flowcharts. Such imaginative tools are rarely available in practice, though, and the programmer is left to digest the raw trace information by himself to determine the path of the control flow for that particular execution of his program.

C. Recent Developments in Debugging Techniques

In the last decade, little change has occurred in the nature of the debugging tools typically available to programmers. DDT and similar systems are still widely used, as are the various simple facilities provided with specific languages such as FORTRAN, PL/1, and INTERLISP. During these years, however, there *has* been some research on debugging that leads in a new direction. This recent work tends to emphasize

- increased participation of the user in guiding the debugging process, supported by the use of graphic displays for higher-bandwidth communication;
- orientation of tools towards the control and data structures of a particular programming language rather than those at the machine-language level; and
- integrated illustration of the relationship between program code and the computational behavior it produces.

The work of Gaines [Gaines, *The Debugging of Computer Programs*, 1969.] was an early contribution in this direction. Satterthwaite's was another [Satterthwaite, *Source Language Debugging Tools*, 1975.], emphasizing post-execution presentations for ALGOL-W. Although not specifically aimed at debugging, Swinehart's work on multiple process environments [Swinehart, *COPILOT*, 1974.] has direct application to this domain. Another example of this new generation of debugging systems is the display system for SP/k recently produced by Yarwood. [Yarwood, *Toward Program Illustration*, 1977.]. This section discusses the three themes that characterize these efforts.

The Use of Display Terminals. Gaines noted with regard to facilities such as DDT that a "difficulty with these systems is that they are limited by the slow speed with which a teletype or typewriter can give the user information. Whereas the programmer using a batch system could get fifteen or twenty pages of trace information for a fairly short run, it would take quite a long time for the same amount of information to be typed out on a typewriter." [Gaines, *The Debugging of Computer Programs*, p. 123.] Teletypes are therefore not useful for the kind of bulk information transfers associated with traditional dump and trace techniques. Even with respect to compute-time, it is much more efficient to write information to files for later printing on a high-speed device than it is to print characters on a teletype. Developments in teletype technology have led to machines with far better characteristics from the user's point of view than those which were available ten years ago had. Nevertheless, even with modern non-contact-printing terminals, the basic rate of information transfer is not much faster than with the old teletypes. It was not until display terminals became practical that significant increases in communication rates became possible. Gaines noted a gain of three orders of magnitude [Gaines, *The Debugging of Computer Programs*, p. 125.], and contemporary equipment probably surpasses even that.

The use of better terminals and display devices in turn affected the design of many aspects of user-oriented computing facilities, such as text-editors, debugging systems, and interactive programming languages. Complete memory dumps have fallen out of use in many environments, giving way to tools emphasizing more interactive interrogation of the state of the machine. Due to the greater speed of modern terminals and display devices, traces can now include much more information during execution than was formerly possible. Even tools originally developed for use in earlier teletype-based systems can be used more extensively than was formerly feasible. Such is the case for INTERLISP's TRACE function, which causes printouts to be produced each time a traced function is called during execution. These printouts include the name of the function, the name and value of each of its arguments, and, when the function finishes, its value. The faster the printing speed of the terminal, the more a programmer can make use of this facility without overly retarding the program's execution.

The increased rate of data transfer, along with the general interactive approach, encourages the user to make fewer changes before testing a program with yet another execution run. In an interactive environment, the time and effort required to obtain information are dramatically less than they were in older batch-mode systems. The vast amounts of information produced by old-style dumps and by complete post-execution traces have been exchanged for a much smaller amount of information selected for its particular relevance and shown to the programmer while the program runs. This advance is not cost free, however; the time the programmer saves is to some extent paid for with computational resources. In a batch system, the programmer tends to spend more time carefully examining his program and the debugging information he had obtained from earlier executions before trying another run. The modern approach encourages a more superficial analysis and more frequent test runs.

Display terminals are faster simply because they are electronic devices, rather than mechanical ones. In addition, they can show information of a qualitatively different type than the character-printing of the teletypes and printers which they mimic. Depending on the particular kind of device, graphs can be drawn, arbitrary halftone pictures displayed, text underlined or shown with varying brightness, and characters printed with different font styles, sizes, and faces. Another significant difference is that the screen represents a single page, and for many devices, once information has "scrolled" off the top of the screen it is gone. This leads designers of display-based facilities to emphasize modifications to the already visible display rather than reprinting of new information following the existing display.

For example, a modified version of DDT called RAID was developed at the Stanford Artificial Intelligence Laboratory. [Petit, *RAID*, 1970.] It can display the contents of several selected memory locations, constantly updating the display as these values change. For the purposes of most interactive debugging activity, the current value of a variable or the contents of a memory location is of primary interest. Display terminals make it possible to continually update the presentation of current values. This kind of dynamically changing display contrasts with the lengthy printouts that result when the values of traced variables are printed each time they change. As Satterthwaite noted, display-based failities such as RAID "have demonstrated that [even] relatively conventional tools are considerably enhanced by increasing the bandwidth of the communication path." [Satterthwaite, *Source Language Debugging Tools*, p. 221.]

Another advantage of display terminals is that information on the screen can be highly organized. Many different pieces of information can be presented in a constantly changing display, but the user need only look at those pieces which are of immediate interest. The PARRY project at the Stanford Artificial Intelligence Laboratory made use of an impressive graphical display of the computation involved in analyzing and responding to an input sentence. [Personal experience; this display facility seems not to have been documented.] The screen was organized into many boxes, each one containing rapidly changing symbolic information showing the state of a particular internal process of PARRY. It was interesting to note while watching this display that the great rate at which new information was presented made it impossible but also undesirable to completely follow the computations. Because the separate processes were each monitored in a different area of the screen, it was very easy to watch one or two of these at a time and ignore the rest.

Thus, one could pay attention to the box in the PARRY display which represented the spelling corrector, say, or the generation module, without being distracted by information about other processes. Shifting attention to a different process required no explicit action, as would usually be the case with traditional debugging tools -- one simply had to look at a different part of the screen. In a different application of multiple screen areas, Swinehart noted the value of what he called "contextual information", especially as presented visually. "This sort of presentation can be made to satisfy the 'continuously accurate' requirement [of interactive programming systems, the subject of his study], without flooding our sensory channels . . . one need not look at everything all the time, and in fact can select what to look at, and when to look at it." [Swinehart, *COPILOT*, p. 11.]

The benefits of display terminals that have been mentioned to this point relate to the flow of information from the machine to the user. If the display terminal is

equipped with facilities such as a pointing device, special function keys, or a light pen, gains in the other direction of the communication flow can be realized. If instead of being required to type out entire commands, the user is able to press only one key, or point with a device at an entry of a menu displayed on the screen, much typing can be saved. In addition to saving the user's time and effort, this also eliminates many typographical errors. Not only do such errors cause frustration and perhaps even result in computational activity damaging to the system's state, but they are also time-consuming to correct. Swinehart pointed out more substantial advantages of such input devices:

> Most of the topics we have discussed have dealt with symbolic terms: with language, its uses and effects. But a remarkable number of things people do are not (at least at the conscious "interface") expressed symbolically at all; they are instead "manipulative" activities. We affect things directly by moving them; we sense them directly by touch, sight or smell.
>
> . . . [While using manipulative, rather than symbolic, devices] the conscious mind is aware only of their effect. This feeling applies especially to those operations which have an immediate and visible effect -- for instance, the movement of cursors or the deletion or movement of text on a display screen.
>
> What we are advocating here is that the way in which such repetitive operations have to be performed be made simple enough that one thinks of them (while doing them) only in terms of their effect. In this way they tend to lose any symbolic meaning and to become practically bodily extensions. [Swinehart, *COPILOT*, p. 12. Note the connection between Swinehart's point and the discussion of quantitative limitations on human information processing found in Chapter 2 of the present work. See also the discussion of human visual information processing in Section B of Chapter 6, below.]

Satterthwaite's debugging facilities for ALGOL-W were batch-oriented; nevertheless, they achieved something like the effect of a graphical display through appropriate paragraphing of text in conjunction with statement execution counts. Yarwood's made the most extensive use of real pictorial presentations of any of the systems mentioned here. He constructed "a system to produce illustrations of the execution of programs in SP/k, a subset of PL/1. These illustrations consist of program text, pictorial 'snapshots' of the program's data, and explanatory messages." [Yarwood, *Toward Program Illustration*, p. 6.] His system, like Satterthwaite's, displayed program text in an appropriately formatted manner, but also took advantage of a software package for generating the graphical aspects of his snapshots and an electrostatic plotter to print them.

Source Language Orientation. The second characteristic of recent work to be discussed in this section is the orientation of debugging tools towards a particular programming language and its constructs. Writing in 1973, Mann complained that "Very respectable tools for debugging higher-level language programs . . . have been proposed, and in a few cases implemented, over the past few years, but none are standard in the more common computer lines." [Mann, "A Survey of Debug Systems," p. 182.] He also emphasized that the majority of programmers have no debugging tools for the

high-level languages which are available to them. [p. 198.] Satterthwaite noted that "As a rule, debugging monitors have been designed as adjuncts of particular object machines (or their assembly languages), not of particular programming languages." [Satterthwaite, *Source Language Debugging Tools*, 1975, p. 22.] Gaines was certainly correct when in 1969 he wrote:

> Debugging systems like DDT have proved to be very valuable debugging tools under the circumstances for which they are appropriate. They are quite suitable for debugging machine language programs, but have the same problems for higher level languages that other debugging aids have had: they cannot give the programmer information in terms of his program. Even if, as is sometimes the case, a symbol and label table is made available by the compiler, the programmer still cannot see his code displayed in appropriate form, or get his variables and arrays displayed with type and dimensioning information automatically taken into account. For example, the programmer cannot specify that "X(3,4)" be displayed and expect that the dimensioning information he supplied in the program will be used to locate the 3,4 element of array X, or that the value for a variable will be typed as integer, real, complex, etc. depending on the type of the variable in the program. [Gaines, *The Debugging of Computer Programs*, pp. 122-23.]

Swinehart investigated multiple process systems having "naming and data allocation facilities which allow multiple recursive or parallel instances of the data environments for procedures, and multiple use of names by scope-qualification." [Swinehart, *COPILOT*, 1974, p. 27.] Multiple processes compound the difficulties which arise from the lack of language-level mechanisms for accessing data and procedural states. Yarwood experimented with ways of pictorially representing high-level data structures such as arrays. [Yarwood, *Toward Program Illustration*, 1977.] In all of the efforts mentioned here (and others like them), there was an overall attempt to provide the user with tools for talking about computational states and operations in the same terms as those in which he wrote the code that produced them. This is a very important theme, and discussion in later chapters will return to it.

The ability to "single step" a program is quite useful for intensive debugging of small pieces of code. Even researchers who specifically consider single stepping often fail to realize that the concept of a single step should be language-based. For instance, although Gaines did address some language-level issues, his debugging system used a machine-level notion of single stepping. [*cf. The Debugging of Computer Programs*, pp. 140-142.] The FORTRAN programmer would like to be able to proceed one FORTRAN statement at a time instead of one machine-language instruction at a time. Swinehart's system was one which did provide a language-level stepping facility. There, the language addressed was SAIL, a dialect of ALGOL.

One difficulty in designing single-stepping facilities for high-level languages is that an appropriate concept of the notion "single step" is not always obvious. In an ALGOL-like language, should the notion be defined in terms of single lines of code, statements which do not contain other statements, or individual operations in the language, such as function calls and arithmetical operators? In an *APL* program, would

a single step be one line of code or one operation? In LISP, would it be a single function call, the evaluation of a single S-expression (one call to EVAL), or the evaluation of an outermost statement of a PROG? It would be useful in language-level debugging systems to provide mechanisms for advancing the computation by each of the notions of "single step" that seem intuitively natural for the language in question. The concept "single stepping" can also be profitably generalized to that of "N stepping" for certain applications, as is done in some systems.

Illustration of the Relationship between Code and Behavior. The final characteristic of recent work to be discussed here is related to the previous one. Traditional debugging tools are oriented towards the direct presentation of information about the sequence of operations and data states resulting from the execution of a program. The previous paragraphs pointed out that with traditional debugging tools, the user cannot phrase requests to the system in terms of the source-language program code. In addition to addressing this shortcoming, recent work has aimed at providing the user with a clear picture of the relationship between the code he wrote and the computational activity it produces. Thus, in Satterthwaite's system, trace information and error reports are presented in the context of printouts showing the relevant pieces of program code. Swinehart's system maintains a pointer to the line of code currently being executed and allows the user various kinds of interactive control based on that notion of context. Yarwood's snapshots intersperse graphical pictures of the states of data structures with printouts of the text of the code affecting those structures.

The illustration of the relationship between program code and the computational activity resulting from the execution of that code is important because program code and computational behavior actually lie in two different conceptual domains. The execution of a program by a computational environment can be viewed as a function which maps each program and the input it receives into a computational sequence [as defined in Chapter 3, Section C.]. More formally, consider a computational environment **E**, including its assemblers, interpreters, compilers, the hardware which interprets the binary code produced by that software, and other less significant facilities. Let **C** be the set of all programs which can be processed by **E**. For a chosen notion of computational behavior, let **B** represent the set of all computational sequences that can be represented in terms of the operations and constructs intrinsic to that particular notion. Let the set of all inputs that can be provided to programs in the computational environment **E** be designated by **D**. Then, **E** can be regarded as a function with domain the Cartesian product **C X D** and range **B**. [The various algebraic properties of this function are interesting to think about, but further discussion would be out of place here.]

Programmers rely on this basic causal relationship between code and behavior. Although the code a programmer writes lies in one conceptual domain, the production of certain corresponding activity in the other domain is intended and anticipated. Likewise, observed computational activity is presumed to be directly caused by some part of the program being executed. The process of debugging depends on the programmer's ability to determine which section of the code is responsible for some undesirable observed behavior. The skills a programmer acquires are thus used to mediate between the domains of code and of behavior. Programming is a *synthetic* activity which focuses primarily on the transformation from the domain of code to that of computational behavior. Determining the source in the code for some aspect of observed computational behavior is an *analytic* activity which moves in the opposite direction between these two domains. Figure 5 illustrates these two domains, and the programmer's mediation between them.

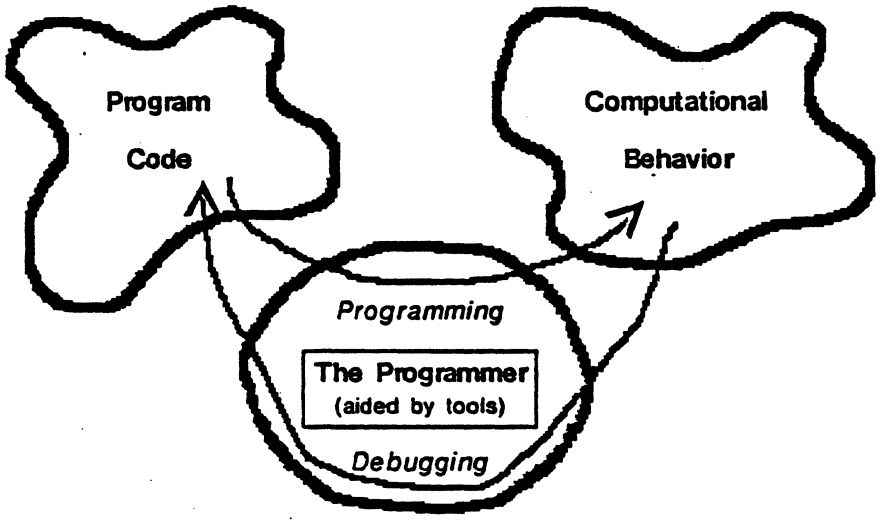


Figure 5. The Domains of Program Code and Behavior

So, to correct a program, the programmer must make an association between aspects of the observed behavior and the parts of the program that seem to have caused them. [*cf.* Aron, *The Program Development Process: Part I*, p. 173 (and preceding) for some discussion of this discovery process in the context of a traditional programming environment.] While establishing this relationship doesn't in itself *correct* the problem, and may in fact obscure more subtle effects not caused by the suspect code, it at least gives the programmer a likely starting point for his efforts at correction. Traditional tools essentially provide information about program behavior in terms of very basic computational elements, such as the value of individual memory cells and machine-level instructions. Given such primitive information, associating the observed

computational behavior with particular pieces of program code is not necessarily straightforward. Also, if the programmer loses track of what part of the code is being executed, it may be quite difficult to reconstruct that context, either during program execution or by examining post-execution printouts. Moreover, the task of finding the causes of erroneous behavior is facilitated by a thorough understanding of *all* the activity produced by execution of the program, not just its suspicious elements. The new techniques for integrating information about computational activity with the actual code of the program help illuminate the causes of the program's behavior. In turn, effective implementation of these tools requires that computational activity be described in terms of the constructs and operations of the programming language in which the code was written.

D. The Act of Debugging

Fundamentally, debugging is the act of (1) observing the behavior of a computer program (or, more precisely, observing the activity of the computer when it executes the program); (2) comparing the actual behavior to the behavior desired of the program; (3) analyzing the cause of variances thereby detected; (4) devising changes to the program that would make it conform more closely to the desired behavior; and (5) altering the program in accordance with those changes. Normally, the act is cyclical: after the program has been modified, the steps are repeated until a sufficient match between desired and observed behavior is obtained. [Note that the production of observable behavior does not require execution of the program on a computer: in simpler situations, hand simulation is often valuable, especially in the earlier stages of debugging.]

It must be emphasized here that the concept of behavior on which debugging is based is that expressed in Section C of Chapter 3; it addresses the procedural steps and internal structures of the program as it performs its computations, not only its overt output behavior. The purpose of this section is to show that although debugging is a five-phased activity, traditional debugging techniques focus almost exclusively on the first phase, the observation of computational activity. From this it follows that in the description and naming of the kinds of aids traditionally provided to assist the programmer in testing and correcting his programs, the term **monitoring** should be preferred over the more commonly used term **debugging**.

Program alteration is usually accomplished through text-editing systems rather than through the use of debugging tools. Some debugging tools permit making temporary patches in the running code, allowing the debugging session to continue, but require that corresponding changes to the source code be made at a later time. This limitation is due to the fact that code is written in a high-level language and compiled

into machine-level instructions before being run. [An advantage of interpretive systems such as LISP and, to an extent, *APL* is that code can be tested and altered at the level of the original source language, after which these alterations can be saved permanently. For example, in INTERLISP, the debugging and editing mechanisms are strongly interrelated: changes made in the course of debugging are equivalent to edits made at the top level, and edited functions can be easily dumped to a file at the end of a session.] Most debugging tools give the user access only to the machine-level code produced by compilation of high-level source code.

Even in those recent systems that trace or illustrate program code at the level of the original language, access for the purpose of *alteration* is either denied or is available only at the machine level. Furthermore, the process of compilation is such that it is would be very difficult to construct a system that permitted the user to make local changes to the source code which could then be recompiled and patched into a set of machine-level instructions. This is because the decisions made during compilation, such as those concerning the allocation of registers for temporary results, make different pieces of the resulting machine-level code interdependent. Consequently, changes to a local section of code may invalidate conditions on which the compilation of succeeding sections had been based.

The fourth phase of debugging activity -- devising the program changes -- is much like the basic act of writing the program. Debugging tools provide little, if any, assistance in this domain. With respect to the third phase -- analyzing the cause of the differences between the observed and desired behaviors -- traditional debugging tools are helpful only in the simplest ways. Except for a few hints about which area of the code or what data structures are involved, the programmer is left on his own to discover the causes of the detected problems. Programming, program modification, and analyzing teleological and conceptual errors are all highly *creative* activities.

Thus, in the third and fourth phases of debugging activity, the programmer operates in a very intuitive mode, depending more on insight and imagination then on rigorous step-by-step analysis. This is partly because such analysis is difficult, but mostly because the effort required to perform a thorough analysis would simply not prove worthwhile. The experienced programmer is often able to recognize immediately, or with only a small amount of investigation, the cause of a problem by proceeding in a clue-guided, exploratory manner. [Mathematical proofs and some detective stories present pictures of methodical analytic thinking which are somewhat misleading. In recent years there has been increased interest in the notion that these kinds of rigorous demonstrations can be produced only after intuition, insight, and recall processes have generated most of the understanding. See, for instance, Koestler, *The Act of Creation*, Chapter VII.] Due to their demands for creativity, the analytic and creative aspects of debugging in a traditional computational environment must be performed almost exclusively by the programmer.

Traditional techniques thus have little applicability in the third, fourth, and fifth phases of debugging activity. Furthermore, there are only a few traditional tools which can contribute to the second phase of debugging -- comparing the observed and expected behavior. [Assertion statements, traps or demons that are set off when certain conditions are met in the computational state, and conditional breakpoints are examples of tools that *can* be used in this aspect of debugging activity.] Thus, although a few facilities provide some assistance in the other phases of debugging activity, especially the second, *the primary focus of traditional debugging tools lies in the domain of behavior observation.* Debugging therefore remains an activity requiring human intelligence.

One pragmatic limitation on the scope of debugging techniques is that computer assistance in the more analytic aspects of debugging activity would require a formal representation of the desired program behavior in terms of entities the machine could monitor and compare with the representation -- function calls, variable values, etc. Research is proceeding in this and other areas relating to Automatic Programming, such as Program Verification and what are sometimes called "programmer's apprentices". [See, for example, Rich and Shrobe, *Initial Report on a LISP Programmer's Apprentice*; Buchanan and Luckham, *On Automating the Construction of Programs*, 1974; Green *et. al.*, *Progress Report on Program-Understanding Systems*, 1974; and Manna and Waldinger, *The Logic of Computer Programming*, 1977. The latter, incidentally, is an excellent tutorial covering most aspects of formal approaches to programming.] This technology has not yet reached the state where it can be applied in debugging facilities for everyday use. As mentioned above, assertion statements -- a major component of the formal approaches -- can be employed both pragmatically as special forms of conditional breakpoints and theoretically in the formal definition of program behavior. The fact of the matter is that under current conditions, for the programmer to produce a complete, formal specification of the behavior the program was intended to manifest would be a task on the same order of difficulty as writing the actual program.

Moreover, the same kinds of errors can arise in such a formal specification as in the original code. In practice, it generally takes trained programmers much less effort to make the comparisons themselves on an *ad hoc* basis than to produce these formal specifications and apply techniques which can make use of them. As for the creative aspects of debugging -- those oriented towards altering the program -- automation awaits substantial advances in the field of Automatic Programming. Nevertheless, progress in the near future may well alter the traditional pragmatic balance that favors clever hacking over careful analysis and mechanical verfication. [See, for instance, Dershowitz and Manna, *The Evolution of Programs*, in which debugging is considered as a special case of program modification in an automatic program development system based on abstract program schema. Sussman's *A Computational Model of Skill Acquisition* is also relevant here.]

This section has shown that traditional debugging tools are directed at the first, and, to a minor extent, the second, phase of a five-phase activity. These tools have changed little since the beginnings of computer programming. Most of the major developments have been with respect to the rate and form of communication about program activity. Primarily, debugging tools are to programs what oscilloscopes are to electronic circuits -- probes. Since these tools are of little use in most phases of debugging activity, a more accurate term for these facilities would be **monitoring tools**, and the latter will be used in subsequent chapters. For debugging facilities to become anything more than probes awaits major developments in the areas of Programming Theory, Automatic Programming, and Program Verification. The traditional emphasis on the nature of the information which monitoring tools give the programmer and the way in which that information is presented has therefore been necessary and appropriate.

In particular, as will be emphasized in later chapters, it is imperative that information about the behavior of a program be presented at the conceptual level at which the program was written and in terms of the constructs and operations of the programming language used. One direction for potential developments in debugging techniques is towards the increased predigestion of information for the user, another theme to be discussed below. Because of the quantitative constraints on human information processing that Chapter 2 discussed, the higher the conceptual level at which information is presented, the more easily it can be comprehended. Monitoring tools which are designed with this fact in view leave more of the programmer's mental resources available for the demanding analytic and creative phases of debugging activity, for which traditional tools are of little use.

Chapter 5

Artificial Intelligence Systems

As was noted in the first chapter, this work focuses on the kinds of complex computational environments produced in the field of Artificial Intelligence. Many difficulties and frustrations arise during a programmer's attempt to understand the behavior of an AI system. This chapter discusses the characteristics of these systems that are the source of this intractability. It is argued that in trying to understand AI systems, it is best to view them not as *programs*, but rather as complex *systems* which are *composed of* programs of a traditional sort. Of particular importance is the effect that these characteristics have on the utility of traditional monitoring tools and techniques.

A. Difficulties in Defining the Behavior Desired of an AI System

It was pointed out in the previous chapter that the process of debugging rests on the programmer's notion of the activity he desires his program to perform. During or after the program's execution, its actual behavior is compared with the behavior desired of it. Ideally, the programmer isolates discrepancies, seeks out their causes, corrects the problems discovered, and tests the program again. In a traditional computational environment, as was indicated in Section C of Chapter 3, program behavior (from the point of view of the programmer) is inherently definable in terms of sequences of operations or of sequences of computational states. The progression of operations that move the computer from state to state is determined by the program code, and there are particular "control paths" that are expected to be followed during program execution. The situation is different in an AI system, and this section discusses some of the reasons for this.

The principle goal of Artificial Intelligence research is to construct computational systems which perform intelligent activity similar to that of humans. That we understand very little of how people do what they do (or why, for that matter) is hardly debatable. The wide variety in academic and non-academic disciplines devoted to increasing our understanding of human activity attests to this fact. What understanding we do have of human activity is at the most general level. Detailed

specification of the mechanics of human activity at whatever level -- biochemical, psychological, etc. -- seems a long way off, if possible at all. Since the underlying structure and causes of human activity cannot at this time be thoroughly and precisely formulated, researchers are not in a position to design computational systems that accurately mimic the mechanisms of human intelligence. They can at best hope to simulate these mechanisms as they are now vaguely and intuitively understood.

In fact, many researchers do not even make *that* attempt. They aim simply to produce behavior which *looks* like human behavior but which may be accomplished through very different mechanisms. Herbert Simon, a key figure of the field and one of its main theoreticians, takes just this stance. He states: "Artificiality connotes perceptual similarity but essential difference, resemblance from without rather than within." [Simon, *The Sciences of the Artificial*, p. 13.] He argues further that "The more we are willing to abstract from the detail of a set of phenomena, the easier it becomes to simulate the phenomena," [p. 16.] and that "the possibility of building a mathematical theory of a system or of simulating that system does not depend on having an adequate micro-theory of the natural laws that govern the system components." [p. 20.]

Simon's view is not shared by all who are interested in Artificial Intelligence research. Nevertheless, even if his position were accepted, it would reflect only on the possibility of constructing a system which exhibited artificial intelligence and on the method of proceeding towards the realization of that goal. It would not be relevant to an understanding of the operation of these systems. One would not expect the low-level activity of a contemporary AI program to parallel that of the mechanisms underlying human intelligence, whether because one shares Simon's point of view or because one would like to copy human mechanisms but realizes they are currently insufficiently well understood for that to be a feasible goal. Consequently, the validity of an AI system's performance can be judged only by comparison of its overt behavior with what we know from our ordinary experience about the overt behavior of humans engaged in performing the same tasks. Formal criteria for the correctness of AI system activity at the level of data structures and program operations are impossible to define.

This has enormous significance for the debugging of AI systems. The traditional conception of debugging presumes that program defects can be discovered by comparing the observed behavior with a conception of the correct behavior. For the purposes of programming and debugging, computational behavior has to be defined in terms of the operations and structures of the program rather than in terms of the overt output behavior it produces [*cf.* Chapter 3, Section C.]. However, because specifying the low-level mechanisms of complex activity is so difficult in general and because so little is known about the mechanisms underlying human intelligence, the behavior desired of

an AI system cannot be well formulated at the program level. Consequently, the traditional notion of program *correctness* as the foundation for debugging activity breaks down. In its place is left only an indirect correspondence between the low-level system activity and the gross characteristics of the human performance which the simulation attempts to capture. This indirect correspondence is mediated by the gross overt characteristics of the AI system's performance -- the output produced by the system as it attempts to understand a story, play chess, or construct mathematical proofs. [These relationships are illustrated in Figure 6.] One's judgment of overt system behavior is qualitative and intuitive and provides no immediate guidance as to what lower-level alterations might be made to improve the system's performance.

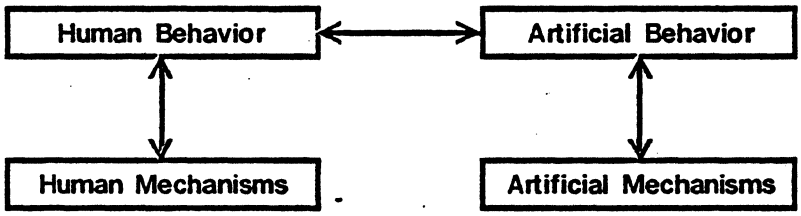


Figure 6. The Relationship between the Mechanisms of Human and Artificial Intelligence

There are several reasons why this qualitative comparison is a poor basis for judging the "correctness" of an AI system's performance. For one, an AI system's failure to perform well may be the result of an insignificant program bug in an otherwise magnificently designed system. Likewise, an impressive performance could be accomplished for a few test cases by a terrible program which could handle only those cases and which embodied no theoretical contribution to the understanding of intelligent activity. Furthermore, it is difficult to formulate suitable evaluation criteria for the human tasks that AI systems are designed to perform. What constitutes a good game of chess, competence in story understanding, or correct interpretation of a visual scene can be only informally described.

Thus, one can evaluate an AI system's performance of such tasks only in an informal, intuitive manner. Because "correctness" is an inapplicable standard against which to measure the high-level performance of these systems, the AI programmer's efforts are directed more toward the *improvement* of system *performance* (as judged informally) than toward the correction of errors in the system's code. Traditional debugging activity gives way to a much more exploratory investigation of system

behavior in an attempt to understand its workings. This is another reason why the term monitoring has been chosen to replace debugging: the latter suggests that there is something *wrong* with the system's behavior that must be corrected, while the former focuses more on *understanding* that behavior. "Programming" and "debugging" in this sort of environment are inherently vague and informal activities compared to the writing and correcting of traditional programs.

B. The Organization of AI Systems

Both the language and structural hierarchies discussed in Section B of Chapter 3 can be extended upward. Representation formalisms developed for Artificial Intelligence research are one way in which this extension can be made. Contemporary examples include systems for representing and processing production-rules [PSG {Newell, "A Theoretical Exploration of Mechanisms for Coding the Stimulus" and "A Theoretical exploration of Mechanisms for Coding the Stimulus".} and the system underlying MYCIN {Shortliffe, *MYCIN* and Davis, Bucanan, and Shortliffe, *"Production Rules as a Representation for a Knowedge-Based Consultation Program"*.} are perhaps the best-known examples.], semantic nets [*e.g.*, Woods, "What's in a Link" and Fikes and Hendrix, *A Network-Based Knowledge Representation and its Natural Deduction System.*], and frame-like structures [See, in particular, Minsky, "A Framework for Representing Knowledge"; Roberts and Goldstein, *The FRL Primer*; and Bobrow and Winograd, "An Overview of KRL, A Knowledge Representation Language".]

Ordinary computer programs are in one sense static descriptions of computational methods; they don't attain their full meaning unless the interpretation process -- execution on a machine -- is taken into account. The same is true for these new representation languages: although they are formal systems for statically describing facts, relationships, and activities, they also appeal to some notion of interpretation. Mechanisms for interpretation may be implicit or explicit and may or may not be implemented. [Unimplemented interpretive mechanisms are nothing new: *APL*, for example, has been used extensively for the description of systems, including hardware, and for its first seven or eight years had no machine implementation at all. {*cf.* Falkoff and Iverson, "The Design of APL," pp. 324 and 333.}] The term representation system is sometimes employed to encompass both the representation language and its interpretive mechanisms, where such exist. The complexity and conceptual power of these representation languages is sufficiently greater than those of high-level programming languages to justify another level in the language hierachy. [Figure 7 shows the extended language hierarchy.] As high-level languages are implemented using assembly or machine code, representation languages are implemented in terms of some high-level programming language, usually some dialect of LISP.

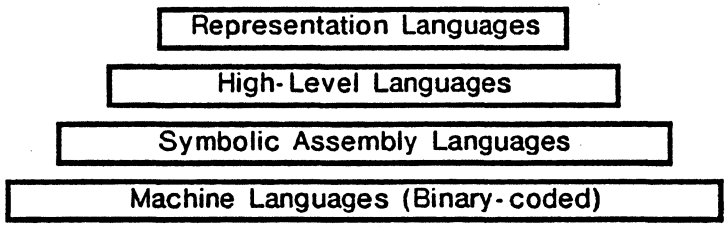


Figure 7. The Extended Language Hierarchy

Large programs are frequently segmented into essentially separate components; for instance, a compiler for a high-level programming language may be divided into a lexical scanner, a syntactic parser, a semantic analyzer, and a code generator. In the most general sense of "computer program", a program can have anywhere from a few to hundreds of thousands of lines of code. However, the size of a system and its complexity are somewhat independent dimensions, and the interactions among the components of even a sophisticated compiler are almost negligible. Although, the notion of complexity has no generally agreed on formal definition, Simon's characterization will serve well for the present purposes:

> Roughly, by a complex system I mean one made up of a large number of parts that interact in a nonsimple way. In such systems, the whole is more than the sum of the parts, not in an ultimate, metaphysical sense, but in the important pragmatic sense that, given the properties of the parts and the laws of their interaction, it is not a trivial matter to infer the properties of the whole. In the face of complexity, an in-principle reductionist may be at the same time a pragmatic holist. [Simon, *The Sciences of the Artificial*, p. 86.]

Complex computational systems are composed of programs, much as programs are themselves composed of subroutines. Complex systems are thus on a level above that of programs in the structural hierarchy. [Figure 8 shows the extended structural hierarchy.] Many representation systems are complex systems in the sense adopted here. The term application systems will be used to refer to systems built using representation systems. Most application systems are complex in the present sense. The term AI systems, explained in the previous section, will be used ambiguously to mean either complex representation systems or complex application systems.

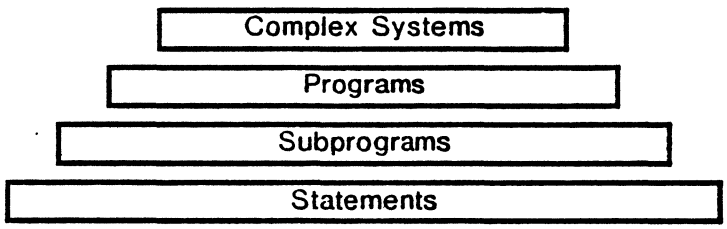


Figure 8. The Extended Structure Hierarchy

AI systems typically stretch the limits of available programming technology, their complexity challenging the abilities of their designers on a variety of fronts. Isolating segments which perform relatively independent functions is a widely used means of organizing these systems in an effort to cope with their complexity. [Although this is characteristic of most implementation efforts in the field, the recent article by Erman and Lesser is one of the few to consider this theme explicitly. {Erman and Lesser, *System Engineering Techniques for Artificial Intelligence Systems, passim*, especially pp. 4-7.}] The value of such decomposition is partly explained through Simon's parable of the two watchmakers [Simon, *The Sciences of the Artificial*, pp. 90-92.], which illustrates an important advantage of hierarchically organized systems -- systems which are composed of subsystems that are themselves constructed from lower-level subsystems, etc. This parable illustrates why these systems have greater resistance to destructive external influences than do complex systems built directly out of low-level parts. [In the parable, one watchmaker attempts to assemble an entire watch from its elementary pieces. A second watchmaker first constructs ten-piece subassemblies, then unites every ten of these into a higher-level subassembly, and so on, until the watch is constructed. The point is that if a disruption forces a watchmaker to put his work down, causing the pieces to fall apart, the first watchmaker will lose his entire work at every interruption, whereas the second will lose at most an incomplete ten-piece assembly. Clearly, under similar conditions, the second watchmaker will successfully assemble many more watches in the same amount of time than will the first.]

An important factor not mentioned in Simon's parable is that such systems are also more robust in their reactions to internal malfunctions, because problems are contained in each part, much as a ship's bulkhead limits flooding to a single compartment and a firedoor retards the spread of a fire to other parts of a building. Both of these advantages were discussed in the simpler context of Chapter 3, Section B ("The Importance of High-Level Programming Constructs"), under the subheading "Reduction of Programming Errors". Although ordinary programs are often designed using decomposition techniques, the relationship among the parts of a complex system and that among the parts of a simpler program are fundamentally different. There is little interaction between the subroutines of an ordinary program. Each one is essentially a separate unit which takes a few pieces of data and performs certain well-defined manipulations on them, finally returning the information thereby produced. Large and complex programming systems tend to be constructed as what Simon calls "nearly decomposable systems": hierarchically organized systems "in which the interactions among the subsystems are weak but not negligible." [Simon, *The Sciences of the Artificial*, p. 100.] Simon continues:

> The main theoretical findings from the approach can be summed up in two propositions: (*a*) in a nearly decomposable system, the short-run behavior of each of the component subsystems is approximately independent of the short-run behavior of the other components [as in an ordinary computer program]; (*b*) in the long run, the behavior of any one of the components depends in only an aggregate way on the behavior of the other components.

He later demonstrates that:

> The fact, then, that many complex systems have a nearly decomposable, hierarchic structure is a major facilitating factor enabling to us [*sic*] understand, to describe, and even to "see" such systems and their parts. Or perhaps the proposition should be put the other way round. If there are important systems in the world that are complex without being hierarchic, they may to a considerable extent escape our observation and our understanding. Analysis of their behavior would involve such detailed knowledge and calculation of the interactions of their elementary parts that it would be beyond our capacities of memory or computation. [Simon, *The Sciences of the Artificial*, p. 108.]

One interesting kind of hierarchical development in Artificial Intelligence programming involves the recognition by some researchers of a set of operations and abilities needed by almost any AI system. Some examples of these are, at various system levels, backtracking routines, a means of representing temporal relationships, mechanisms for considering several competing hypotheses, and syntactic parsers. Not only is it wasteful for new versions of these to be created from scratch in each system, but when a system is large and complex, building the lower levels can completely overwhelm the resources available for the project as a whole. It is important for the progress of the research for which these systems are constructed that work start near the level at which new techniques and facilities must be developed.

The KRL project at Stanford University and Xerox Palo Alto Research Center [of which more later] is one project in which the value of hierarchical development is explicitly recognized. One of its goals is the construction of a hierarchy of facilities needed by AI systems designed to perform reasoning tasks, specifically those involved in the processing of linguistic phenomena. Bobrow and Winograd wrote:

> There is currently no suitable base on which to build sophisticated systems and theories of language understanding. A complete understander system demands the integration of a number of complex components, each resting on ones below. . . . Current systems, even the best ones, often resemble a house of cards. The researchers are interested in the higher levels, and try to build up the minimum of supporting props at the lower levels. The standardly available (such as LISP, QLISP, CONNIVER, production systems, etc.) are at a low enough level that many middle layers must be built up in an ad hoc way for each project. The result is an extremely fragile structure, which may reach impressive heights, but collapses immediately if swayed in the slightest from the specific domain (often even the specific examples) for which it was built. [Bobrow and Winograd, *An Overview of KRL, A Knowledge Representation Language*, p. 1.]

In that report, Bobrow and Winograd propose to develop a hiearchy of an "underlying computer programming language and environment"; "basic strategies" such as for search and reasoning; "common sense domains" to deal with "time, events and states, plans and motivations, actions and causes, . . ."; "linguistic domains" of "syntax and parsing strategies, morphological and lexical analysis, discourse structures, . . ."; and "task domains" for such things as "travel arrangements, medical diagnosis, story analysis, etc." [Bobrow and Winograd, *An Overview of KRL, A Knowledge Representation Language*, Figure 1, p. 1.] Not only would such a system be a nearly-decomposable complex system, but it would exhibit these characteristics at the level on which they interact with their users.

C. Data in an AI System

The monitoring of system behavior is strongly affected by the hierarchical organization, complexity, and near-decomposability of AI systems. Effective monitoring in such an environment must be based on an approach significantly different than that underlying the traditional tools and techniques designed for program-level debugging. As discussed in Section C of Chapter 4, some recent work on debugging techniques has recognized the value of tools which approach computational activity in terms of the constructs and operations inherent in the programming language being used. That theme becomes even more important in the context of complex systems. Certainly, machine-level tools are even less appropriate to such systems than they are for high-level language programs. Even tools oriented toward the program level are inadequate in a complex environment. One no more wants to view KRL-1 units in terms of the underlying INTERLISP data structures than to investigate INTERLISP constructs with DDT or a machine-language program by watching the console lights or with oscilloscopes attached to key circuits. In pursuing this theme, discussion once again will begin by considering problems associated with observation of data states and then turn to those relating to computational processes.

In ordinary programming, choosing the data structures with which to represent the entities to be manipulated during execution is an important aspect of the programmer's task. The word "structure" in the term "data structure" is suggestive in the context of the present discussion. A programming language provides several means for data representation, such as integer-valued variables, arrays, strings, and lists. The choice among the available constructs is made on the basis of the potential contribution of each *structure* to clarity, brevity, efficiency, taking into account the nature of the operations that are likely to be most frequently performed on the structure. For example, where many numbers must be stored in sequential order and accessed frequently by sequential position, arrays are a natural choice. A list-structured representation would be inefficient since the program would have to step a pointer through the list until the desired location had been reached. However, lists can be altered more flexibly than can arrays, and if accessing is to be performed by a sequential search on a key, then lists and arrays might be equivalent with respect to speed. A program-level monitoring tool would display information in a manner appropriate to the particular structure of the data: arrays in tabular form, recursive structures in a nested printout, and so on.

Languages with record facilities, such as PASCAL and INTERLISP, encourage the user to design his own data structures. Some control might be provided over the way the structure is actually represented in terms of the basic constructs available through

the language, but the main purpose of this kind of facility is to make the program more readable. By allowing pieces of record structures to be accessed through field names, record facilities encourage the programmer to substitute for arcane accessing sequences expressions which are more meaningful to him, constructed from operations on the basic available data structures. A simple case in INTERLISP might be the replacement of an expression such as

```
(CADADR (ASSOC 'TAX EMPLOYEE))
```

with one that looked something like

```
EMPLOYEE:TAX:FEDERAL.
```

This mimics to some extent the use of macro facilities in program code. Similarly, some languages, such as MESA [Mitchell, Maybury, and Sweet, *Mesa Language Manual*, Chapter 3.] and INTERLISP [Teitelman, *INTERLISP Reference Manual*, Section 23.] permit the declaration of new data types, defined in terms of the ones already present in the language.

A potential disadvantage of using language declaration features is that the standard debugging tools will have little information about how the new structures should be printed out in response to user requests. To the extent that representation systems make only *moderate* use of the record and data type facilities of the languages in which they are implemented, the intensity of this problem is the same as for any ordinary sophisticated use of those facilities. Unfortunately, the data structures implemented in a representation language often involve *unusually extensive use* of declaration facilities in the underlying implementation language. Such structures may contain a large amount of highly organized information and may be interconnected in ways that simply cannot be comprehended by looking at their parts using program-level tools. Often, a user who was not involved in the original design of the system could not interpret the implementation-level structures which represent the system-level constructs. It would not be sufficient to devise suitable formatted printouts for each of the system-level representation structures along with their declarations, even if that were convenient. The problem is of a larger magnitude.

In general, the user of a representation system would not want to see an entire structure, a KRL-1 unit or process, for example. Even if the implementation-level details were disguised through an appropriate printout, there might be so much information associated with each structure that the user would have trouble comprehending its significance. Moreover, he might have a hard time extracting from the mass of information the particular aspect of the structure in which he was originally interested. In fact, since these structures tend to be intricately cross-linked, developing a method for showing all the information in a finite printout could be a difficult problem. The underlying failure of the traditional approach is a deep one: in

the standard break-and-examine paradigm, questions about computational states are answered simply by showing the user the data and leaving him to determine its characteristics.

Not only are the higher-level constructs of a representation system more intricate than typical program-level structures, but the notion of data is fundamentally different in a complex AI system. Facts may be provisional, there may be many competing versions of the same hypothesis, and information may be only partially known. [An important area of AI research involves what is often termed "reasoning with uncertain knowledge". "Certainty factors" are attached to hypotheses, and "answers" to a problem may be phrased in terms of several conflicting conclusions with their associated certainty factors.] At this level, greater demands are put on a monitoring system than those involved in responding to a traditional question asking for the value of a variable or a request to print out some portion of an array. The user will want to ask questions about the *nature* of the data, not just about its raw *state*. To respond appropriately, a monitoring system for a representation language must not only have built-in knowledge about how to display that language's basic kinds of data entities, but must know something about the *semantics* of their structures. Only then could a system provide high-level assistance to the user who asks to see "all active processes" or "those hypotheses having a certainty factor greater than .8" or "all the events caused by this demon".

D. The Flow of Computation in an AI System

Perhaps the most significant difference between AI systems and traditional computer programs is with respect to the flow of computational activity. Much of the user's attempt to understand system behavior involves following the flow of control as computation proceeds, with particular attention paid to places where control decisions are made. In traditional languages, control decisions are made at clearly identifiable locations in the code -- conditional statements and their derivatives. Control flows in a coherent direction, whether based on linear progression through the code (as in algorithmic languages), recursive evaluation (as in list-processing languages), or in the sequential processing of messages (as in Smalltalk [Goldberg and Kay, *Smalltalk-72 Instruction Manual.*]). Following the flow of computation is basically a question of determining *which* of the essentially pre-determined paths through the code is actually being followed.

In AI systems, however, control decisions are not localized at particular statements in the code, and the flow of control is often highly incoherent. The "intelligence" of an AI system is due in part to their creating their own computational paths based on internal goals and information. Confronted with such autonomous

behavior, the AI programmer often has the feeling that he doesn't understand what the system is doing. In this sort of environment, the monitoring task involves much more than just determining *which* control path is being followed. The system user puts considerable effort into discovering *what control paths the program has created* in its self-generating computations. Moreover, the imposition of new control structures on top of those of the implementation language puts system-level operations beyond the reach of lower-level monitoring tools.

The Imposition of New Control Regimes. The first generation of new Artificial Intelligence programming languages attempted to provide some of the standard control techniques of heuristic programming as built-in facilities. [A valuable survey of these languages, their goals, and their achievements is found in Bobrow and Raphael, *New Programming Languages for AI Research.*] These languages included SAIL [Swinehart and Sproull, *SAIL*; Feldman, *et al.*, "Recent Developments in SAIL".], PLANNER [Hewitt, *Description and Theoretical Analysis of PLANNER* and Sussman and Winograd, *MICRO-PLANNER Reference Manual.*], CONNIVER [Sussman and McDermott, *Why Conniving is Better than Planning*; McDermott and Sussman, *The CONNIVER Reference Manual.*], QA4 [Rulifson *et al.*, *QA4.*], and the spaghetti-stack extension to INTERLISP [Bobrow and Wegbreit, "A Model for Control Structures for Artificial Intelligence Programming Languages".] The utility of backtracking, pattern-directed invocation of procedures, coroutines and generators, and (simulated) parallel processing was sufficiently demonstrated in various AI programming efforts that facilities implementing them were included in these new languages. Although these are not quite complex systems in the sense described above, by providing alternative control mechanisms these languages do cause serious problems that hinder the user's attempt to understand his system's activity. In true complex hierarchical systems, this problem is, of course, even more severe.

With the utilization of alternative control mechanisms, the handles provided by the standard language-level debugging tools become almost useless. Tools within the standard break-and-examine paradigm must be based on the notion of execution sequencing and control transfer that was built into the language for which they are designed. Basic control-oriented debugging facilities provide ways of specifying places where a break is to occur or a trace produced. What the natural junctures are for such specifications depends on the control regime of the language. Thus, DDT permits breaking on an individual instruction, and INTERLISP's BREAK package facilitates stopping at calls to particular functions or at expressions within the code.

However, the particular notion of control juncture embedded in a higher-level control scheme may be entirely different than that of the implementation language on top of which it is imposed. For instance, the basic cycle of a simple production rule system might be to search through memory collecting those rules which are relevant to

the current state, order them appropriately, and apply each of their actions. The user of a disease diagnosis system based on production rules might want to suspend the computation each time any rule involving Pneumonia is tested for applicability. Language-level tools provide no direct, natural way of interrupting such high-level processes. Instead, the user must determine the names of the functions and variables which implement the system-level operations, then apply language-level tools to these low-level components. Individual function calls and the values of program variables are at too specific a level of detail for the user of a complex system to be concerned with. The system-level programmer wants to observe high-level descriptions of system activity, not the step by step performance of their internal implementation mechanisms.

Consider for example a system that features pattern-directed invocation of processes. There, the decision about what to do next depends on the match of a pattern against each member of a set of assertions. Using the tools of the programming language in which the system is implemented, there would be no way to specify the equivalent of "break when a match with assertion A0047 is found", the analogue to "break when function FOO is called". Another difficulty in such a system, or in one based on backtracking mechanisms, is that the "next" activity performed is not "next" in any sense detectable by a traditional debugging tool. The use of the standard break-and-examine approach is disrupted because it requires that breaks be defined in terms of points in the code, not in terms of overall system activity.

Thus, the flow of control in these new kinds of systems escapes the reach of traditional debugging mechanisms. Of course, since the systems are implemented in a programming language, there is a place in *some* code where the decisions are made and activities invoked. *But those places are in principle out of reach to the user of the language and should remain so*, just as the basic decisions made within LISP's EVAL are inaccessible to the user. Monitoring techniques must be applied at the level at which the user conceptualizes and writes his code, and whenever a system, even one as relatively primitive as these early Artificial Intelligence languages, provides a new level, the monitoring tools previously available are evaded.

Non-Local Control Decisions. Some complex systems are designed so that several processes can affect the same data structure. These processes may run in parallel or in indeterminate order. Their code may have been written by several different people. These characteristics make it infeasible to monitor system activity by tracking the particular places in the code where interesting changes to the data structures are made. [The organization of the HEARSAY system developed at Carnegie-Mellon University is a prime example of this kind of complex multiprocess structuring. {See, for instance, Erman and Lesser, "A Multi-Level

Organization for Problem Solving Using Many Diverse Cooperating Sources of Knowledge"; Lesser and Erman, "A Retrospective View on the Hearsay-II Architecture"; and Erman and Lesser, *System Engineering Techniques for Artificial Intelligence Systems.*}] In this kind of system, the focus of attention becomes the changed data structures themselves, rather than the parts of the code that cause the changes.

In KRL-1, for example, there are several mechanisms for making control decisions. One involves the use of demon-like triggers. Because these are stored declaratively as parts of Units, their execution is not a local event in the process that causes their firing. Another mechanism relies on signals issued as part of the execution of particular pieces of code but whose responses are catalogued globally. A third mechanism is based on agenda-oriented process invocation and suspension. In this kind of complex environment the question "What is happening?" is not appropriately answered by pointing to a particular piece of code. Instead, it requires description in terms of concepts higher than those intrinsic to algorithmically oriented traditional programming languages. As is the case for data structures, such description requires that the monitoring system have some knowledge of the semantics of the process activity.

The Non-Linearity of Control. Given some piece of AI system code which contains consecutive statements A, B, and C and which runs as one of several parallel processes, it cannot be assumed that no activity has occurred between the execution of statement B and start of statement C. Consequently, monitoring these processes involves more than finding out which statement of the code is being executed, and direct traces of what piece of code or even what process is being executed would be too confusing to be of much use. Again, it is not simply a matter of printing out the information in a correct form. The essentially linear model of traditional computation is inappropriate to a multi-processing or backtracking system. Non-linear methods of showing the user what the system is doing are required. Displays must be organized conceptually, rather than temporally. Examples of this would be a printout of all events affecting a certain hypothesis, or all the activity involved with a certain process, or all rules that contributed to some goal state.

The non-linearity of system activity also affects the way in which the user is required to specify what he wants to see. When decisions about what to do next are made on an "intelligent" basis, the user can never be sure that a particular piece of code will ever be executed, or a particular process ever resumed. This contrasts with traditional programs, where the programmer has an idea of at least the *kinds* of computational sequences that can occur; in a complex system, the variety is too great to be managed directly. A major part of the monitoring task must then be devoted to

finding out what the system is doing at the moment, what made it decide to do that, and what it is attempting to achieve by doing it. The programmer in a traditional environment is concerned only with the first of these, in order to discover *which* sequence is being executed. The non-linearity of system activity thus excludes the domain of possible sequences from the language of monitoring requests.

Pragmatic considerations. Finally, a pragmatic consideration conditions the kind of monitoring activity that is appropriate in a complex environment. AI systems are notoriously large and slow. Because so much varied activity may occur in a particular execution, it is often tempting to narrow one's monitoring focus during one execution, then rerun the system with the same parameter settings, input data, goal specifications, etc., but with different monitoring requests in effect. For instance, each run may serve to show the development of one out of several hypotheses. Because so much activity is performed by a complex system, it is expensive to run. At the same time, precisely *because* so much activity is performed, the programmer cannot monitor it all simultaneously (or even, given the intricacy and instability of such systems, be confident of reproducing it in subsequent runs). A monitoring facility for a complex system must deal with this apparent paradox. It must provide a means of viewing system activity from a variety of perspectives which does not require multiple re-executions of the system for that purpose alone.

Conclusion. This chapter has shown the importance of recognizing a structural level above that of programs and a linguistic level above that of programming languages. A point to be emphasized is that the conceptual distance between programs and systems and the distance between programming and representation languages are in some respects greater than the distances separating lower levels of the two hierarchies. Monitoring techniques and tools must be different *in kind* -- not simply in degree -- from the traditional ones that have been in use since the earliest days of computer programming. Representation systems in the field of Artificial Intelligence are *complex systems* in a fundamental sense and not merely large programs. They are *composed of* programs. Each of those programs can be treated as such and debugged at the program and programming-language levels, but the nontrivial interaction among these separate parts of the system produces complex phenomena. The nature of these new kinds of phenomena calls for a non-traditional approach to the design of tools and techniques that aid the programmer in dealing with them. The second half of this work presents one such approach to monitoring the computational behavior of a complex system and demonstrates its generality and feasibility.

Part Two

The Design of Monitoring Facilities for AI Systems

Chapter 6

An Approach to Monitoring the Activity of Complex Systems

Chapter 4 concluded by showing that traditional debugging tools are directed predominantly at enabling the user to obtain information about the behavior of his program. They serve essentially as probes into the space of data states and control paths traced out by the program's execution. The preceding chapter discussed the characteristics of AI systems that make it difficult for users to understand their behavior. This chapter puts forth an approach to monitoring system activity designed to deal with those difficulties. The approach maintains the traditional emphasis on observation; this is based on the belief that progress in the development of aids for the other aspects of debugging activity (especially the comparison of observed and expected behaviors) must await extensive advances in the theory of programming languages and in the representation of processes. [*cf.* Chapter 4, Section D and Chapter 10, Section A.] A part of this proposed approach involves extensions to traditional tools that would make them useful in these new kinds of environments. As indicated above, however, more is required than simple improvement. Accordingly, this approach incorporates new techniques that go beyond the traditional break-and-examine method for debugging programs.

The approach has three major components. First, the upgrading of traditional monitoring tools is required, so that they will deal with system level entities, instead of with pieces of the program and data structures by which the AI system is implemented. A more important step in this direction is the development of monitoring tools which will answer the user's questions *about* system activity instead of simply showing it to him directly (meta-monitoring). The second aspect of the approach emphasizes the importance of graphical, non-symbolic presentation of information. The third thrust is to take into account in the design of monitoring systems the difference between the static investigation of discrete comptuational states and the dynamic observation of active processes. The chapter following this one presents a generalized method of describing system activity which complements and grounds the basic approach put forth here.

A. System-Level Monitoring Tools

It is important that AI representation systems be opaque with respect to their implementation. Ideally, the user should not have to know anything at all about the programming language underlying the system. Even where the user is familiar with that language, knowledge about how particular system-level entities and processes are implemented should neither be required nor assumed. It should be possible for the user to investigate the behavior of an AI system without his having any knowledge of system function names, variable names, or details of the code. This section examines ways in which AI systems and monitoring tools for them can be designed in accordance with this policy of insulation.

Printing and Editing System-Level Structures. Section C of the previous chapter described the intricacy commonly exhibited by data structures in an AI system. Entities such as competing hypotheses, nodes of a semantic net, and frame-like objects are often structured in a multi-levelled, interconnected way. As a result, even where the implementation language or its debugging system provides facilities for formatted printing of data structures, the printouts may be unintelligible. For example, one node of a list structure X may point to the list structure Y, but some node of Y may point to X. When one prints out either X or Y, the circular structure will be printed indefinitely, unless special printing routines are available for handling such structures. Even then, either the user or the system must recognize the circularity of the structure so that the appropriate special routines get invoked. Other mechanisms used in the implementation of system structures that make them unintelligible to the user include various kinds of encoded indirection (such as numerical indices, hash links, or pointers to file positions), non-standard representations of characters or numbers, and unlabelled substructures interpreted by the system according to their position in the larger structure.

Problems arise also from the creation of new data types. Language-level printing mechanisms which have been programmed to handle the basic data structures of the language are not prepared to print newly defined data types perspicuously. At the very least, AI system builders should provide formatted printing facilities for the system data structures they implement which mimic and extend those available for data structures in the implementation language. Not only does this give the user a tool to fall back on when more sophisticated ones either are not available or do not meet some specific need, but these printing facilities are invaluable during the construction of the AI system itself. Such facilities are simple enough to implement, yet one system after another is constructed without anyone involved bothering to do so. As a result, much time and effort is wasted puzzling over confusing printouts.

Although the emphasis in the present work is on the observation of system structures and activity, it is worth noting here that these considerations about system-level printing apply equally well to system-level editing. The manipulation of system structures at the implementation level requires an even better understanding of their format than does observation. A user having some familiarity with the implementation of a structure can *partially* comprehend the bizarre printouts resulting from the use of traditional tools in a complex environment. However, disastrous results can ensue from such a user's attempts to alter system structures through their underlying representations. Undocumented conventions, the necessity of maintaining cross-references between parts of structures, and misleading printouts (due, for example, to non-printing characters in a string or objects which print the same despite being distinct) trap the adventurous user. Moreover, the system user may find that ordinary editing facilities do not allow him to make the changes he desires, as a result of some non-standard usage of language features within the structure.

System-Level Breakpointing. Formatted printing of data structures is one form that the "examine" part of the traditional break-and-examine approach to debugging can take. The final section of the previous chapter explained a variety of ways in which the new control structures of an AI system evade the grasp of "break" mechanisms designed for the implementation language. Another minimal step the implementers of an AI system must take is to provide mechanisms for specifying breakpoints in terms of system-level concepts. For example, the basic control cycle of a production-rule system is to decide what rules are applicable in the current context, order them in some way, apply them, and repeat the process. A natural place to interrupt the computation would be before the application of some rule, but it is difficult to accomplish this with instruction- and function-oriented debugging mechanisms designed for the implementation language. Another example of data-dependent control decisions is the pattern-directed invocation of procedures in languages like PLANNER and CONNIVER.

The problem is not just one of access. Although the available implementation level tools may be in principle *sufficient* for specifying all sorts of breakpoints and conditions, experience shows that their use is abandoned when any but the simplest invocations would be required to achieve the desired effects. Using traditional facilities, the expressions that would be required to designate system-level breakpoints are likely to be complicated. Furthermore, to construct these invocations, the user would have to be familiar with the implementation code and the names of the relevant variables. To elevate the traditional break-and-examine paradigm to the system level, mechanisms for specifying breaks in terms of system-level concepts must be provided.

Then, the user would be able to type to an INTERLISP-based system, for instance,

```
PAUSEAT(RULE0037)
```

instead of

```
BREAK0(CONTROL1 (AFTER MAINLOOP)
                (EQ (fetch (RULE NAME) of NXTRUL) 'RULE0037))
```

The extension of break-and-examine type tools to the system level should be an integral part of the system-building effort. In part, this is because efficient implementation depends on their being carefully coordinated with system code. Then, too, it is the system builders who are familiar with implementation details and can track changes in them as they are made.

Meta-Monitoring. The approach to monitoring presented here emphasizes techniques which go beyond the traditional break-and-examine paradigm. It is quite feasible for these new techniques to be embodied in a separate monitoring facility that could be constructed by someone having only a rudimentary understanding of the AI system implementation. [In fact, there is no reason why the monitoring facilities should have to be implemented in the same language as the AI system for which they are designed.] Such a monitoring facility can even run independently, acting as an interface between the user and the AI system. An independent monitor must acquire detailed information about the activity of the AI system as it executes and have some "knowledge" about its significance. The next chapter exhibits a principled method for producing such information. Section C of Chapter 8 discusses mechanisms for effecting this kind of separation between an AI system and its monitor. Here, a concept termed meta-monitoring is introduced as a central part of the approach. It is proposed that: (1) the user should be able to ask questions *about* computational states and processes rather than merely obtaining raw information about their parts; and (2) an advanced monitoring system should be able to notice basic patterns in system objects and answer simple kinds of user questions concerning these patterns.

With a machine-level debugging system such as DDT, the user can have the contents of individual memory cells printed out. For multi-celled data structures, such as arrays and lists, this is quite an awkward way to examine parts of the data state. Even if a debugging tool is capable of showing the contents of many memory cells at once, it may not be able to indicate the *structure* of multi-celled objects. Arrays are generally represented internally as one-dimensional sequences of locations. The array may be multidimensional, though, and it will not in general be obvious how to map a given set of multidimensional coordinates to a position in the linear sequence of memory cells. In addition, that mapping scheme is at too low a level for even the traditional programmer to concern himself with. There is a similar problem for linked

structures such as lists. The programmer is forced to manually trace through the linkages one step at a time, interpreting the contents of each memory location to decide what location to look at next. Source-level debugging tools [including the INTERLISP BREAK package {Teitelman, *INTERLISP Reference Manual*, Section 15.}, BAIL {Reiser, *BAIL: A Debugger for SAIL*.}, and various FORTRAN systems developed over the years.] are of some help in this respect, in that they allow the user to specify elements he wants printed out with expressions using the syntax of the programming language itself and the identifier names from the original program. As systems become more complicated, however, element by element investigation of a compound structure, and even formatted printing of the entire structure, becomes unwieldy.

Frequently, even in a traditional environment, the programmer is interested only in the gross characteristics of some structure. Using the standardly available tools he would nevertheless have to first print out the entire structure then survey the mass of data thereby produced from the viewpoint of the characteristics with which he is concerned. Consider an array in which all but one element had the value 0. Not only would it be computationally inefficient to print all the elements in such a case, but for the programmer to scan the expanse of numbers looking for a pattern and the exceptions to it is a waste of his time and limited mental resources. This is another example of the inverse relationship between information and meaning that was discussed in the section on "The Significance of High-Level Programming Constructs" above [Chapter 3, Section B.]. The repetitious printing of the same value provides very little significant information and obscures, instead of illuminating, the highly meaningful special state of the array. It would be preferable if the user could ask "What is the state of the array" and get back an answer like "still initialized", "all zeros", or "mostly 1's, with the following exceptions . . .", or a standard printout when one of these simple patterns was not perceived by the system. [The automated compilation of low-level information was investigated only tentatively for the program level in the course of the present work, so this example is illustrative rather than indicative of any tool actually implemented. (*Cf.* Chapter 10, Section A.)] In such simple yet frequently-occurring cases, the monitoring system could compile low-level bits of information into a less detailed, more meaningful response.

Such a facility would be useful at the program level. At the system-level, the intricacy of data structures makes this more than a mere convenience. Consider a system-level structure such as a KRL-1 Process. The user might want to know the status of the Process or how much of its resources it has left. Using traditional tools, the user would either print out the entire structure and look through the resulting details, as in the above example of arrays, or type in some expression in the implementation language that would access only the relevant part of the data structure. Both sorts of

action require an understanding of implementation-level details that would conflict with the principle of system opacity put forth at the start of this section. Appropriately formatted printing would help here, but many kinds of high-level structures contain so much information that the user would be overwhelmed if it were presented all at once. For example, the Current Hypothesis of a medical diagnosis system may print as an entire page of text, whereas the user wants only to know its name or the system's estimate of how well it accounts for the observed symptoms.

One method of handling large amounts of detail is to isolate for each structure one or two fundamental features and print only those, pending user requests for more detail. For instance, in one of the experimental systems developed for this work, initial displays of KRL-1 Agenda structures showed only their type and the names of the Processes on their queues. For a Hypothesis representing a medical diagnosis, the name of the suspected disease and the estimate of how well the Hypothesis accounts for known symptoms would be sufficient. Furthermore, the name of an object is often all the programmer needs to see, as printouts frequently serve only to *identify which one* of a set of known objects a particular one is. When the user does want further details, it is imperative that he be able to request them selectively. Even a tool which did no more than expand the standard SHOW X debugging command to a SHOW ASPECT OF STRUCTURE command would be a significant advance for complex systems. [The user will be familiar with the names and features of the various aspects of each kind of structure, since this is the level at which he programs. It is the implementation form of the structure which should be concealed from the user, along with those of its aspects not currently of interest.] The examples of the ninth chapter (particularly the one concerning the KRL-1 trigger-based system) extensively illustrate the use of meta-monitoring to obtain answers to substantial questions without thereby loosing overwhelming torrents of low-level details.

B. The Power of Analogical Visual Presentations

In the manner of Chapter 2, "Some Aspects of Human Information Processing", this section presents some psychological and physiological facts about human perception. These show that mechanisms for comprehending analogic information in graphical form are in some respects vastly more powerful than those underlying the processing of abstract, symbolic information.

The Primacy of Vision in Humans. The basic facts of human physiology and psychology reveal a strong bias in the human organism towards visual processing. "More than half of the several million nerve fibers that connect the brain with other parts of the body go to the eyes." [Wooldridge, *Mechanical Man*, p. 42.] There are about 1,200,000 neurons that carry information from the retina of each eye to the brain; by

contrast, there are only 50,000 neurons connecting the ear and the brain. [Lilly, *The Mind of the Dolphin*, p. 88, where it is noted that in dolphins, as is appropriate for their underwater environment, the ear predominates over the eye by about the same proportion.] Assuming that the channel capacities of the neurological circuits associated with the various senses are similar, this means that the human organism receives as much information about the world through the eyes as through all other sensory modalities combined.

Pre-Cortical Processing of Sensory Information. Modern neurophysiological research has shown that sensory information is not channeled directly to the brain; instead, neural networks process the raw data to extract significant features (in vertebrates, this occurs in several stages). [*cf.* Wooldridge, *The Machinery of the Brain*, Chapter 3; Gregory, *Eye and Brain*, Chapter 5.] "Since, for most animals, a large fraction of all the information that comes in from the outside world is visual, it is in the visual sensory systems that the most opportunity and necessity exist for processing and rearranging the input data before the transmission to the brain." [Wooldridge, *The Machinery of the Brain*, p. 38.] In a classic study of the frog's visual system [Lettvin *et. al.*, "What the Frog's Eye Tells the Frog's Brain," 1959.], it was found that "there are four separate operations [performed] on the image in the frog's eye. Each has its result transmitted [to the brain] by a particular group of fibers . . . , and they are all nearly independent of the general illumination. The operations are *(1) sustained-contrast detection; (2) net convexity detection; (3) moving-edge detection; and (4) net dimming detection.*" [Lettvin *et. al.*, "What the Frog's Eye Tells the Frog's Brain," p. 239.]

Another classic work [Hubel and Wiesel, "Receptive Fields, Binocular Interaction, and Functional Architecture in the Cat's Visual Cortex," 1962.] demonstrated the existence in the cat's brain of line detectors which were sensitive to the angles of the lines. It showed further that apparently, "much of the rearrangement of the raw optical data that occurs in the visual system of the cat is carried out before the signals become immersed in the deeper layers of the cortex." [Wooldridge, *The Machinery of the Brain*, p. 52.] This summary of recent research was offered by one psychologist:

> The picture that has emerged from the results of an incresingly large number of experiments in psychology, physiology, neurology, and the communication sciences implies that the nervous system performs substantial alterations of the physical image received by the sense organs. These transformations extract information about color, enhance contours, determine size and direction of movement of visual images, extract the pitch and loudness of acoustical images, and determine the spatial and temporal relationships of visual and acoustical signals. These transformations are of great use to the nervous system because they simplify tremendously the information that must be trasnmitted to higher-level analyzing systems. This, in turn, simplifies the job of analyzing the sensory inputs. Transformations lose information, however, for aspects of the signal that are combined with one another at one level of processing cannot be separated at higher levels. [Norman, *Memory and Attention*, p. 39.]

Note once again the importance of the increase in *meaning* achieved by filtering and through organization processes that reduce *information*.

The Power of Visually-Based Analogical Comprehension. On the psychological level, the power exhibited by visually-based, analogically-oriented comprehension processes is impressive when compared to that of more abstract modes of conceptualization in the same context. Much information can be grasped "at a glance" which, if it were in a non-visual form, would require more time and mental effort for processing. Visual aids are commonly used for various kinds of information presentation: in advertising, classrooms, professional talks, and so on. The proverbial thousand words is probably a good estimate of the information contained in a picture of moderate size and detail, and the significance of the picture can be understood much more quickly than the equivalent words. [The distinction here is between analogic information based on shapes, colors, brightnesses, spatial relations, etc., and information represented by abstract symbols such as letters and numbers. Symbolic information can of course be presented visually; it is not so much the *mode* of presentation as the *nature* of the information itself that is of concern here.] While abstract symbols such as words may be recognized visually, understanding their *significance* requires a different kind of processing. In addition, the vocabulary of vision is in some ways more powerful than that of symbolic systems, including language. As Arnheim points out,

> The visual medium is so enormously superior because it offers structural equivalents to all characteristics of objects, events, relations. The variety of available visual shapes is as great as that of possible speech sounds, but what matters is that they can be organized according to readily definable patterns, of which the geometrical shapes are the most tangible illustration. The principal virtue of the visual medium is that of representing shapes in two-dimensional and three-dimensional space, as compared with the one-dimensional sequence of verbal language. This polydimensional space not only yields good thought models of physical objects or events, it also represents isomorphically the dimensions needed for theoretical reasoning. [Arnheim, *Visual Thinking*, p. 232.]

It seems likely that the greater efficiency in the processing of analogic information depends to a large extent on the pre-cortical organization of optical data, by which manipulations are performed which otherwise (for symbolic information) would have to be accomplished by the higher thought centers of the brain. Even from the standpoint of evolutionary theory, it is natural that the human organism be able to process visual information more efficiently than the symbolic information which the species has only recently acquired the ability to handle. The world of the human organism is primarily a physical one, and the nature of that world is grasped primarily through visual modes. Visible objects or pictures facilitate the organization of perception and thought by acting in effect as mnemonic "hooks" on which mental images can be "hung." [This is why it is often effective for a speaker to show a slide outlining what he is saying, even though the information displayed thereon is redundant. Similarly, the scribbles some lecturers spread over a chalkboard aid the listener to organize and comprehend the talk -- even when those scribbles are undecipherable -- simply by acting as physical loci for concepts and relationships.] Finally, visually presented analogic information is stored very efficiently in Short-

Term Memory, permitting greater comprehension within the fixed bounds on the contents of Immediate Attention. The following statement is suggestive:

> *Visual organization* is an extremely efficient chunking method. A closed curve that can be remembered topologically contains an infinite amount of information about individual points. One thing experimenters usually do with data is plot it on a graph or in a table, to organize it visually. Concepts become clear only when the overall structure of individual pieces of information can be grasped. [Smith, *PYGMALION*, p. 10; the entire first chapter of that report is devoted to a discussion of Short Term Memory, conscious thought, and mental imagery which although not concerned with debugging, relates in interesting ways to various themes in the present work.]

A good example of a visual communication tool is the use of pictorial displays of numerical quantities (as with bargraphs and pie-charts) to illustrate large amounts of data and relationships among its various pieces. The human ability to almost instantaneously form visual "gestalts" -- organized perceptions of the meaning of entire scenes -- accounts for much of the efficacy of this sort of device. Also important in this respect is the earlier discussion about system compilation of lower-level pieces of information into higher conceptual structures to save the user's mental resources for more important and more challenging tasks. There are a variety of typical computational structures and activities which are amenable to this sort of compilation, with the resulting information well-suited to graphical displays. Those activities which can be monitored according to the linear progression of a single quantity are good candidates for such treatment. These include such fundamentally quantitative kinds of information as how far an iteration has progressed, how much of the original resources of a process remain (according to either a single resource dimension or some function combining several dimensions), and how much free storage is left in the system.

Moreover, "how far" and "how much" questions appeal inherently to a physical, not abstract, reality, and are best answered in terms that reflect their implicit metaphor. One device used for such purposes as charity drive publicity is a good example of an appropriate visual representation of progress: a drawing of a thermometer (or something to that effect) is filled in from the bottom up as the drive moves towards its stated goals. Most people are not terribly effective in estimating numerical relationships. The advantage of this kind of analogical representation is that the estimation is performed by our visual hardware, which, though not particularly well-suited for precise measurements, is quite proficient at making rough approximations. Graphical displays communicate the essential features of the situation, in immediately comprehensible visual form. Not only do they make unecessary the many awkward numerical manipulations that would have been required to arrive at the same fact that is compiled into the graphical representation, but the resulting abstract conceptualizations would not have as much cognitive significance as the more easily *visualized* picture.

The Use of Graphical Displays for Analogical Presentations. The approach to monitoring proposed here recognizes the power of visual communication of pre-compiled analogic information. The use of display terminals in debugging systems has been the subject of some experimentation in recent years. [See Chapter 4, Section D.] However, the traditional symbolic mode of information presentation has been generally retained. The features of display terminals that made them attractive to systems designers were

- greater speed of these devices compared with those of hardcopy terminals;
- flexibility provided through their abilitity to make random modifications to the screen, allowing the updating of information or the imposition of spatial organization on the display; and
- highlighting of text with a cursor, underlining, or changes in light intensity.

Rarely have display terminals been used for analogical presentations in a debugging system. [Yarwood's recent work is somewhat of an exception. His system illustrated the state of data structures with a partially analogical graphical display. {Yarwood, *Toward Program Illustration*, 1977.} Also, though not concerned with debugging, the graphically-oriented family of programming languages called AMBIT manipulated structures such as lists and graphs analogically. {Christensen, "An Introduction to AMBIT/L, a Diagrammatic Language for List Processing" and "An Example of the Manipulation of Directed graphs in the AMBIT/B Programming Language".}]

The use of analogical graphical displays can contribute greatly to the communication power of a monitoring system. Both the quantity and quality of the information the user needs concerning the behavior of his complex system make the use of displays almost a necessity, not the luxury it has always seemed. This second part of the present approach to monitoring emphasizes the analogical presentation of information, not just in purely analogical displays, but also in the integration of symbolic and analogic information in the same act of communication. Chapter 8 describes some particular tools constructed in accordance with this aspect of the approach and discusses these themes further.

C. Discrete States and Dynamic Processes

> "A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it."
> [The First Law of Mentat, from Frank Herbert's *Dune*, p. 32.]

The traditional conception of debugging focuses on capturing the state of the computation at some particular instant. This is true of both non-interactive facilities, such as trace packages included as part of the support software for a programming language, and for interactive systems, such as DDT and INTERLISP's BREAK package. The "state of the computation" can be divided into the data state and the process state. At the machine level, the data state encompasses the contents of all memory locations and accumulators and the process state is specified by the contents of the processor registers (in particular the Instruction Register). As discussed in Chapter 3, higher level concepts of state will be associated with higher levels of the language hierarchy. Debugging almost never requires complete determination of the data state, even when dumps of the entire contents of memory are used. In the traditional break-and-examine approach to debugging, the programmer acquires, at certain points in the computation, that information which appears at the moment to be of particular significance. The programmer's aim is to construct, through partial investigation of the data and process states, a general picture of the computation and how it is progressing.

Although the data state is for the most part directly accessible to the user through standard mechanisms for manipulating data structures, the process state is usually stored implicitly in conventional locations which are difficult to access and decipher. For instance, in the DEC implementation of FORTRAN IV, subroutine jumps involve the storage of a return address and some flag bits in the first location of the subroutine to which control is being transferred. Even if the symbol table produced by the compiler could be used to deduce which routine contained a given address, determining the sequence of function calls that led to a particular call would still involve the awkward task of piecing together the information stored in these special locations. INTERLISP is somewhat more helpful in this regard. Although the INTERLISP stack is not a data structure that can be manipulated directly by the user, functions are provided for viewing and modifying certain aspects of it. Thus at break-time, the user can invoke special printing facilities which show the sequence of function calls leading up to the one that is broken (including both system and user functions) and, if requested, the arguments and their values for each of these calls. When the programmer cannot directly investigate the process state, he must try to follow the execution as it proceeds through the program code. Trace tools assist the programmer in this effort by notifying him when designated points in the code have been reached and, perhaps, what

decisions were made at key branch points.

Thus, using reports obtained at a series of discrete points in computational time, the programmer intuits features of the overall computational flow. The distinction between the process state as it stands at some point and the flow of computation over the course of the execution is important. The latter exists as *a pattern in time*, whereas the former is merely *a state at some instant.* The pattern can be *described* or *portrayed* but it is not itself accessible to a language user or his code; only static entities can be created, destroyed, altered, examined, etc., not patterns in time. This partially explains why the programmer is left on his own to construct an informal picture of the flow from static information. Even where the process *state* is represented, its history may not be. [Furthermore, there are some theoretical limits on the level of detail to which activity can be described by the mechanism producing it. For instance, the philosopher Karl Popper pointed out that a computer could never completely describe what it does, because describing what it did would itself have to be included in the description of what it did next, so that the machine could never catch up with itself. See Popper, "Indeterminism in Quantum Physics and in Classical Physics," Part II, p. 174.]

Although the system cannot be expected to represent its own activity completely, there are other related ways in which it could aid the user. For instance, experience has shown that as one executes a program over and over again while debugging it, one becomes familiar with the patterns of its printouts (including trace information). These patterns can be visual, not symbolic. For example, one may learn that the printing of many short lines and an occasional long line is characteristic of the program's behavior in one part of the program and that the appearance of lines of medium length separated by two blank ones indicates the program is nearing its termination. It is the gross characteristics of the printouts which are noticed in this phenomenon: line lengths; the frequency of printouts; the relative proportion of numbers and words appearing; whether the information is sentence-like or tabular; and so on. In learning about a program's patterns of behavior, one also acquires a "feel" about its rate of execution. Then, by judging the rate at which the familiar patterns proceed (and given the current load on the computer), one can judge *intuitively* how much time remains before some expected event occurs (completion of the execution, for instance). It also becomes possible to detect differences in behavior (the result of some edits, perhaps) by noticing that the patterns of printouts are not quite the same as those to which one had become accustomed.

In the early days of computers, a programmer could find these patterns by watching the console lights. A primitive technique for showing progress that is still commonly used is to print a mark on the paper each time a key point in the main

program loop is reached -- each time a new record is about to be processed in an accounting program, for example. The analogical display of such patterns in dynamic graphical form would be another example of "precompilation" of low level pieces of information into a higher conceptual structure, with an added dimension of temporality. It is premature, given the state of research into the representation of processes and the semantics of computer programs, to attempt to construct systems that decide which temporal patterns are interesting or how to show them to the user. Graphical tools *can* be provided, however, which give the user a way to view patterns which he knows are important and can tell the system how to detect.

In addition to facilities for obtaining descriptions of discrete states and isolated bits of information concerning the flow of the computation (as in traces), there are a few commonly used ways of illustrating the *structure* of a program. The most notable of these is the flowchart, another technique used since the earliest days of programming [and yet another technique first suggested in Goldstine and von Neumann, "Planning and Coding Problems for an Electronic Computing Instrument," 1947-48.]. A flow chart diagrams the steps that are to be performed in the computation and, for every conditional in the program, indicates the path which control is to follow for each possible outcome. Thus, flowcharts are pictures of the structure of programs, and in this respect they are quite similar to circuit diagrams and architectural blueprints of plumbing systems. Although these kinds of diagrams show all possible paths through the system, they indicate very little about the system's actual *behavior*. That a particular test is always true, that a particular segment of code is never called, or that most of the compute time is taken up by a certain loop cannot be shown by a flowchart. The diagram of the plumbing is not the same as the flow patterns of the liquids in an actual working system. The volume, pressure, and nature of the liquids flowing through each section of pipe over time is information that can only be hinted at in the schematic diagram. *Static information*, even where it is concerned with the overall structure of the system, *does not fully capture the dynamic patterns* of actual system behavior.

Even though description and representation are important factors here, from the point of view of program monitoring the issue goes beyond that. The *perception* of dynamic processes depends on different kinds of information than that which is presented in static pictures of discrete states. There is a fundamental difference between the experiencing of a motion picture and the experiencing of snapshots taken of successive states. Although in actuality, movie films are nothing more than sequences of individual snapshots, each one only slightly different from the one preceeding it, a dynamic effect is produced when they are projected in so quick a succession that they are not separately perceived. When the sequence of snapshots is

shown in this manner, human visual processing mechanisms are unable to absorb the full details of each snapshot. Organic processing strategies dictate that most attention be devoted to the changes occurring from one snapshot to another.

Additionally, in that the rapid succession of snapshots in a motion picture simulates the continuous action characteristic of real-world scenes, the natural change-oriented interpretative mechanisms in the human brain/mind are brought into play. In contrast, the examination and comparison of individual snapshots would probably occur in the higher centers of the brain/mind. These, due to their generality and more symbolic orientation, are much less efficient than the organic special purpose brain hardware and mind software. Finally, a further deficiency in static pictures, described by Arnheim, is significant in the present context:

> The usual illustrations in textbooks and on the blackboard help to make a problem visible, but they also freeze it at one phase of the range to which the proposition refers. Therefore, they tempt the student to mistake accidental circumstances for essential ones. The solution is not to leave out illustrations but either to produce mobile models, for instance, by means of film animation, or, at least, to use immobile illustrations in such a way that the student realizes which of their dimensions are variables. [Arnheim, *Visual Thinking*, p. 182.]

In sum, there is a qualitative difference between what can be communicated through "snapshots" of discrete states and "movies" of active processes. Although it is possible to derive information about dynamic processes from facts collected about discrete states, this is a tedious process which in most circumstances achieves little success. It is important, especially in dealing with complex systems, that monitoring tools demonstrate the *activity* of a system, instead of just giving the user a way to obtain individual facts about particular states of the machine. Monitoring must involve more than sequential snapshots, whether taken under program control or by the user at a break. *User perception and understanding of system activity are best supported by monitoring techniques which are active and continuous in nature.* These will almost necessarily involve analogically-oriented visual displays.

The previous section concluded with the remark that the use of display terminals in experimental debugging systems has been limited predominantly to more sophisticated presentation of symbolic information. As rare as analogically-oriented facilities for debugging are, truly dynamic ones are even scarcer. Virtually non-existent are systems which combine both an analogic and a dynamic orientation. In the few dynamic systems constructed in recent years, it is *symbolic* information which is communicated dynamically, not analogic information. [EXDAMS is probably the best representative of a dynamically- and symbolically-oriented system for monitoring computational activity. {Balzer, "EXDAMS -- EXtendable Debugging and Monitoring System," 1969.}] Recognition of the importance of dynamic presentation for both symbolic and analogic information is the third thrust of the approach to monitoring put forth in this chapter.

Chapter 7

Describing the Activity of a Complex System

The previous chapter presented an approach to monitoring the behavior of complex systems. This chapter proposes a method for producing descriptions of the activity of any computational system. The method is based on the primacy in each system of a small set of fundamental structures and operations. The succeeding chapter describes some tools for monitoring systems that were implemented in the course of this research using the facilities of a recently created display-based programming environment. The approach of the previous chapter, the description generation method described in the present chapter, and the graphical tools discussed in the next chapter have been experimentally combined to produce several monitoring facilities. These experiments have shown that based on this generalized scheme, monitoring facilities for any particular representation system can be produced with little effort. Of course, not all possible graphical tools were constructed and not all ramifications of the general approach to monitoring were explored. The point of the experiments was to show the validity of the basic approach and the feasibility of producing a generalized monitoring system which, to be applicable to any particular complex system, requires only a modest implementation of the description method described in this chapter.

A. The Fundamentals of Computational Systems

When one looks at a particular AI representation system, it will often be apparent what kinds of entities and activities are generically interesting to the user. For instance, rule-based systems are attractive because they feature a simple, straightforward control scheme and constrained forms of operations that are easily dealt with by people who are not experts in computer programming. When a user wants to monitor the behavior of his rule-based system, a fundamental piece of information that is almost always of primary significance is the answer to the question: "Which rule is currently being considered by the system?" In a system based on difference reduction, such as GPS [See, for instance, Ernst and Newell, *GPS*, 1969; and Newell and Simon, "GPS, A Program that Simulates Human Thought," 1963.], the user would normally be interested in some very basic kinds of entities: he would want to know the state that has been currently, though tentatively, attained; the current goal state; the operation

being applied to transform the current state towards the goal state; and an estimate of the distance between the two states, according to a metric designated for the particular task domain. In a KRL-1 system, the monitoring of process activity would involve showing the current Process and the state of the current Agenda. [Agendas include lists of Processes and mechanisms for choosing the next one to run. Processes contain, among other information, descriptions of goals, resources, and methods.]

Although many AI systems make no commitment to particular methods of declarative representation, when they do implement some scheme, this too becomes a source of system-relative standard items of interest. In such a system, KRL-1 being a good example, accessing, modification, and comparison of data elements are done not in the underlying implementation programming language but through facilities in the representation system itself (Seek, Describe, Match, etc.). In this context it is important that observation of the data state be performed in terms of the activity of these facilities, rather than at the level of the implementation code. In addition, monitoring would naturally be based on the elements of the well-defined format which the system provides for declarative information -- slots in a frame system, for instance.

One way for a monitoring system to be applicable to a variety of programming or representation systems is by operating at a lower level which is shared by all the systems. For instance, DDT works at the machine-language level and can therefore be used to debug both FORTRAN and SAIL programs (among others), as long as these are executed on one of the Digital Equipment Corporation computers that support DDT. In a like manner, since INTERLISP is such a favorite of the AI community, its BREAK package is a debugging facility that is used across almost the entire spectrum of AI systems. For a *high*-level monitoring facility to be generally applicable requires that it be designed in a schematic way. Taking into account the user's needs for information about the behavior of a complex system and the approach to this monitoring activity that was presented in the previous chapter, a generalized monitor can be constructed which would require only modest additional structure and information to make it applicable to some particular system. Many of the experimental aspects of the research reported on here involved the development of such generalized monitoring facilities.

An example of this kind of approach at the programming level was EXDAMS [Balzer, "EXDAMS -- EXtendable Debugging and Monitoring System," 1969.], which provided mechanisms for interrogating and replaying a history of all the significant actions performed in the course of a program's computation. To produce that history, language-specific routines had to be added to the run-time environment and the original code modified by another program so that these routines would be called at appropriate times. However, the facilities in the EXDAMS system that made use of the

history tape were essentially independent of the source language used. While a complete record of the program-level computations of an AI system would be too costly to generate and would contain too much information to be effectively processed, this basic separation of the monitoring system from the details of the language system it is monitoring proves to be a productive and flexible approach. What emerges both from theoretical analysis and from consideration of typical representation systems is that the monitoring which system users wish to perform is intrinsically directed at the fundamental structures and basic operations of the system. These system fundamentals are thus an appropriate focus for the process of gathering monitoring information.

By examination of some of the major approaches to AI system organization currently in use, the following schematization of system fundamentals was developed. The first distinction recognized in the schematization is that between static information structures and active operations. Although this seems related to the traditional distinction between data and code, it actually rests on substantially different ground. Here, the contrast is between stored information and computational activity. The latter does not *exist*, as do both data and code, but rather *happens*. Both traditional data structures and program code (in the general sense that includes such things as production rules and KRL Triggers) are static structures. Other static structures include entities involved in the representation of computational states (such as process descriptions like those of KRL-1) and index associations relating two chunks of information (including hash-coded arrays and system name tables). These four classes of fundamental static structures -- Data, Program, Process, and Association, along with their associated active operations, form a generalized schematization of system fundamentals, as illustrated in Figure 9.

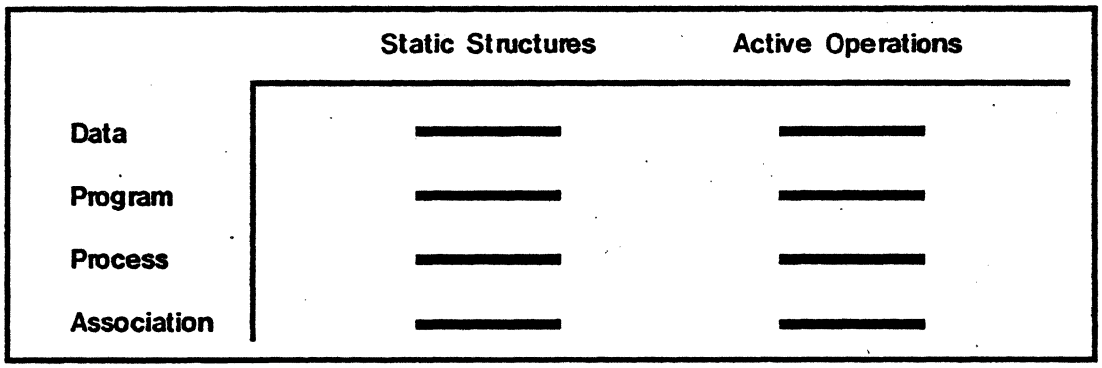


Figure 9. A Generalized Schematization of System Fundamentals

There are six basic operations common to structures of all four groups: **Create, Copy, Destroy, Modify, Access** (of parts), and **Show**. In addition, there are certain operations unique to each group. For data structures, there are the operations of **Compare** and **Combine**. The latter is something of a catch-all group, specific to each particular data structure. It includes, for example, arithmetical operations on numbers, concatenation of strings, appending of lists, logical operators on truth values, and union, intersection, and complementing of sets. Combine operations often modify one or more of the operands, or create a new object. Compare operations may be more complex than simple tests with yes-or-no results -- for instance, the comparison of two patterns may produce information about their differences. The operation special to programs is that of **Interpretation**. Processes are manipulated through the operations **Start, Terminate, Suspend**, and **Resume**. The unique operation for association structures is **Retrieve**. Figure 10 summarizes these fundamental operations.

Common: Create, Copy, Destroy, Modify, Access, Show			
Data	**Program**	**Process**	**Association**
Compare Combine	Interpret	Start Terminate Suspend Resume	Retrieve

Figure 10. The Fundamental Operations of Computational Systems

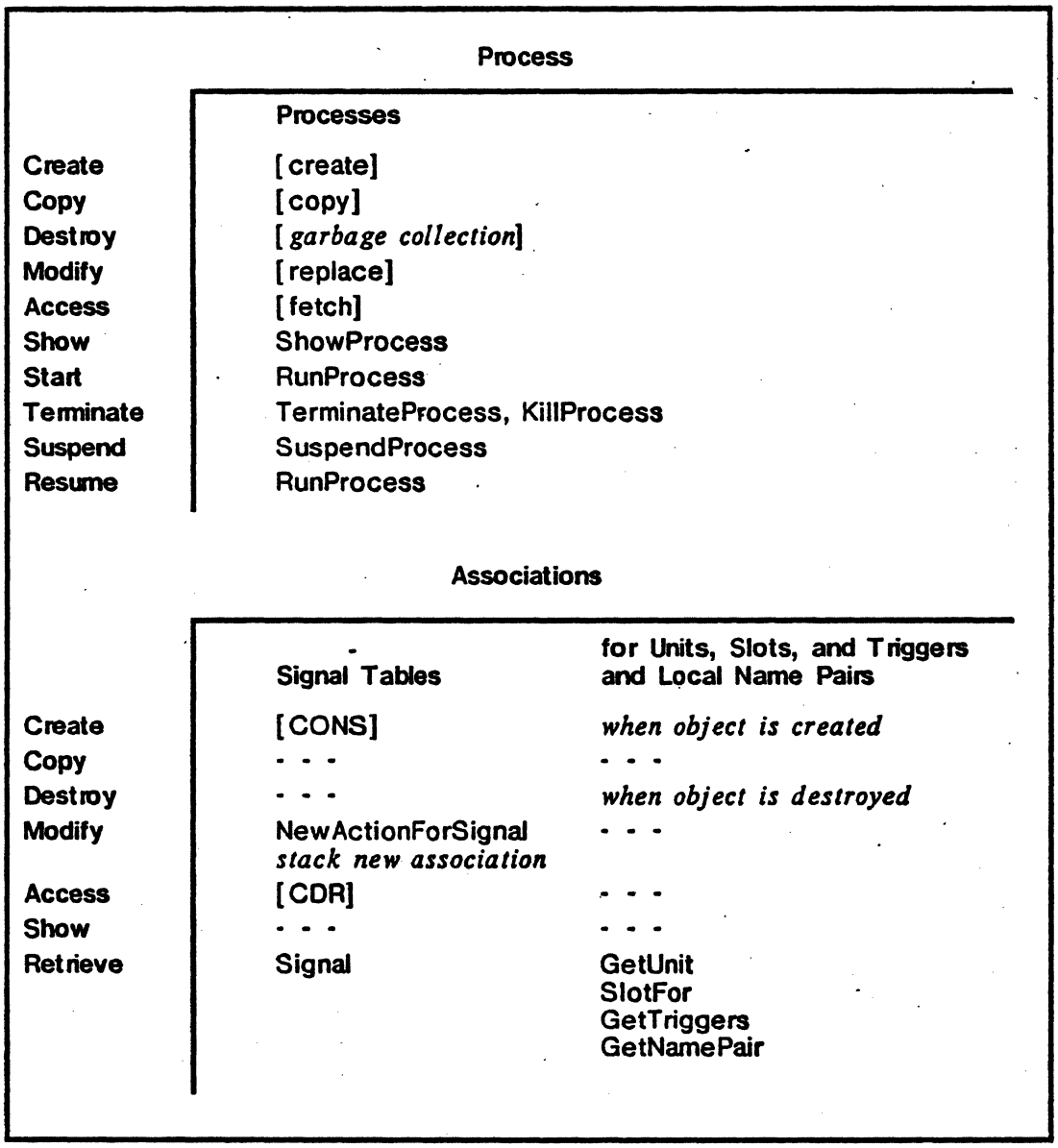
System-level structures in all four groups are not always included, nor is every possibile operation for every static structure in the system likely to be implemented at the system-level. In many cases a system resorts to the implementation-language level for many of the particular structures or operations. For instance, production-rule systems such as the one on which MYCIN is based make little commitment towards a representation for data-objects; these are structured and manipulated directly in INTERLISP, the underlying implementation language. The Show operation is particularly subject to omission, and application of the scheme should include the construction of appropriate Show operations for each fundamental static structure in the system. Figure 11 (a and b) shows how the schematization would look for KRL-1. The same schematization can be developed for traditional programming environments, although this is interesting more from a theoretical standpoint than from a practical one. Figure 12 (a and b) shows how the schematization would look for INTERLISP. [Some details and qualifications have been omitted from these charts for the sake of clarity.]

	Data	
	Units	**Descriptors**
Create	*the KRL reader*, NewUnit	*the KRL reader*, KrlCreate
Copy	- - -	KrlCopy
Destroy	- - -	- - -
Modify	Describe	Describe
Access	Seek	Seek
Show	PPU, PPMU	PPK, PPDESCR
Compare	Match	Match
Combine	- - -	- - -

	Program	
	Triggers	**Process expessions and functions**
Create	*Unit and Descriptor input*	[*various*]
Copy	- - -	[*various*]
Destroy	- - -	[replace]
Modify	[*the editor*]	[*the editor*]
Access	- - -	- - -
Show	PPU	ShowProcess
Interpret	FireTrigger	RunProcess

key: [] means performed on the implementation level (INTERLISP). Words in italics are descriptions, rather than function names.

Figure 11a. KRL-1 System Fundamentals



Key: [] means performed on the implementation level (INTERLISP). Words in italics are descriptions, rather than function names.

Figure 11b. KRL-1 System Fundamentals (cont.)

Datatypes

	Lists	Litatoms	Small integers	Other Numbers	Strings	Arrays
Create	CONS	READ MKATOM	- - -	READ	MKSTRING	ARRAY
Copy	COPY	- - -	- - -	COPY	CONCAT	COPYALL
Destroy	garbage col.	garbage col.	- - -	garbage col.	garbage col.	garbage col.
Modify	RPLACA RPLACD	SETATOMVAL PUTD, MOVD SETPROPLIST	- - -	SETN	RPLSTRING	SETA SETD
Access	CAR CDR	GETATOMVAL GETD, MOVD GETPROPLIST	- - -	- - -	SUBSTRING NCHARS	ELT ELTD ARRAYSIZE
Show	PRINx PP	. . . . PRIN1	PRINx	PRINx	PRINx	HPRINT
Compare	EQ EQUAL	EQ	EQ	EQ EQUAL	EQ STREQUAL	EQ EQUALALL
Combine	APPEND* NCONC# INTERSECTION* UNION*	- - -	arith. ops.	arith. ops.*	CONCAT*	- - -

Program (Functions)

Create	PUTDEF; DEFINE(Q)
Copy	cannot copy compiled code; if non-compiled, then COPYDEF, or MOVD with flag T
Destroy	DELDEF; garbage collection
Modify	EDITDEF; the editor
Access	- - -
Show	PP
Interpret	EVAL, APPLY

- - - not applicable (doesn't or can't happen)
. . . . could be there, but isn't
* creates a new object
\# modifies an old object

Figure 12a. INTERLISP Fundamental Structures

Process (Stack Frames)

Create	EVAL/APPLY (internal), ENVEVAL/ENVAPPLY and variations
Copy	COPYSTK
Destroy	garbage collection
Modify	SETSTKARG, SETSTKARGNAME
Access	STKNAME, STKNTHNAME, VARIABLES, STKARGS
Show	BAKTRACE shows designated portions of the stack
Start	started when created
Terminate	RETURN in PROG code; exit from function code; RETFROM and its variations
Suspend	retain handle on stack frame via stack pointer and then terminate it
Resume	RETTO

Associations

	Atom Properties	Variable Bindings	Assoc List Pairs	Hash Array Entries
Create	PUTPROP* ADDPROP*	LAMBDA args PROG vars	PUTASSOC*	PUTHASH*
Copy	- - -	- - -	- - -	- - -
Destroy	REMPROP	- - -	<NIL>	<NIL>
Modify	PUTPROP* ADDPROP* CHANGEPROP	SETSTKARG SETATOMVAL SET(Q)	PUTASSOC*	PUTHASH*
Access	- - -	- - -	CDR	- - -
Show	- - -	- - -	- - -	- - -
Retrieve	GETPROP	GETATOMVAL STKARG reference	(S)ASSOC	GETHASH

- - - not applicable (doesn't or can't happen)
* if there is an association, it is replaced (modified); otherwise a new one is created
<val> means the association is destroyed by modifying the value of the association to make it be this marked one.

Figure 12b. INTERLISP Fundamental Structures (cont.)

A further aspect of this schematization is the recognition that for each fundamental operation there is a natural set of primary facts which could be displayed or recorded each time the operation is invoked. Consider a KRL-1 Trigger, such as the one contained in the Unit shown in Figure 13. [This unit was taken from an actual KRL-1 application -- a trip planning system developed by the Knowledge Based Assistance project at Xerox Palo Alto Research Center.] When a KRL-1 Trigger is fired, the relevant facts would include the Prototype and Slot to which the Trigger is attached; the particular "Anchor" (piece of description) which was the ground for this particular firing of the Trigger; the program code contained in the Trigger description; the activity caused by the evaluation of this code; and the time the triggering occurred. [To attempt a brief explanation of what all this means would be futile; for more detail, see Bobrow and Winograd, *An Overview of KRL, A Knowledge Representation Language.*]

```
# TripActivity↑1 -- An activity on a day of a trip.
                                1: FurtherSpecified(TripPart)
    self:↑3                     3: Trigger(WhenIdentified, '(InitializeNewInstance))
    startTime:↑4  A Time        4: Trigger(BeforeFilled, '(IncorporateAct))
    endTime:↑2    A Time        2: Trigger(BeforeFilled, '(IncorporateAct))
    description:  A String
    tripActArea:  A TripActArea -- The corresponding Trip Day Form area.
    tripDay:↑5    A Trip Day -- The trip day that this is an activity on.
                                5: Trigger(WhenFilled, '(TestForIncorporateAct))
```

Figure 13. A KRL-1 Unit with Triggers

Consider also the application of a production rule such as the one in Figure 14. [This is from the MYCIN system, an experimental medical diaguosis system developed at Stanford University. {Shortliffe, *MYCIN -- A Rule-Based Computer Program for Advising Physicians Regarding Antimicrobial Therapy Selection.*}] The items of interest here are the antecedent of the rule; whether the rule "succeeds" or "fails"; if it fails, the clause of the antecedent that caused it to fail; if it succeeds, the actions taken as a result (mostly changes to data structures, but possibly process-related ones too); and perhaps some information about the context in which the rule was applied. By looking at examples such as these, it has been found that most operations have approximately five to ten such primary aspects, and that only a small amount of experimentation is required to determine a reasonable set for each fundamental in a system.

```
Premise     ($AND (SAME CNTXT SITE (ONEOF BLOOD CSF))
                   (SAME CNTXT MORPH ROD)
                   (SAME CNTXT GRAM GRAMNEG)
                   (SAME CNTXT GU-MANIP))

Action      (CONCLUDE* CNTXT IDENT TALLY
                   '((PSEUDOMONAS-AERUGINOSA 360)
                     (PROTEUS-MIRABILIS 180)
                     (SERRATIA 100)
                     (ENTEROBACTER 180)
                     (KLEBSIELLA-PNEUMONIAE 180)))
```

Figure 14. A MYCIN Production Rule

The notion of current entitites, which is relevant to many standard ways of organizing computational activity, is yet another part of this scheme. For instance, in the traversal of a tree, an important aspect of a system-level description of the computational state is the node currently being processed. In KRL-1, Processes on some Agenda may be suspended by themselves or by other Processes and resumed later by the Agenda's Scheduler. Here, which is the current Process is an important part of the computational state. Other examples include the rule being applied in a production-rule system and the current goal (perhaps as part of a tree of goals and sub-goals) in a goal-driven process. Current entities, in some descriptions of computational state, are not necessarily true system fundamentals: they may simply be the value of some variable or a part of some data structure, and they may not even be represented explicitly. For example, in the MYCIN system, there is always a current goal, although the system itself does not recognize this explicitly, representing it implicitly instead as the values of the variables CNTXT and PARM. These indicate the feature and object currently being sought, such as the IDENTITY of ORGANISM-2.

B. Events

The generalized scheme presented in the previous section provides an approach to deciding *what* information should be recorded or displayed for fundamental operations. In addition, the proposal asserts that the invocation of each fundamental operation, and in some cases the completion of any associated activity, are natural points *when* information should be recorded or displayed. For KRL-1, a fairly complete picture of system-level activity would be obtained by monitoring the invocation of a subset of the structure-operation pairs: each creation or destruction of a Process, Unit, or Descriptor; each change in the status of a Process (NEW, RUNNING, SUSPENDED, or DEAD); and each firing of a Trigger or signalling of a Signal. This section introduces a kind of structure, termed an Event, that aims at objectifying such aspects of system activity.

Description of System Activity. Events reflect the computational activity of an AI system. They embody a record of the fundamental operations performed on fundamental structures in the course of system execution. The dynamic processes of a computational system cannot be captured in static structures. [*cf.* Chapter 6, Section C.] Therefore, Events do not completely *reproduce* a computation, but merely *describe* them. Moreover, an Event record is incomplete with respect both to what actions are included and the amount of detail reported. Thus, Events describe the fundamental activity of a computational system in high-level terms.

Consider for example the prototype Unit shown in Figure 13 above and an individual Unit described in its terms. When the tripDay Slot of the individual Unit is filled, the WhenFilled Trigger attached to the tripDay Slot of the prototype is fired. This causes the LISP expression (TestForIncorporateAct), a function with no arguments, to be evaluated. A *complete* record of the activity associated with the Trigger's firing would have to include all side effects of that evaluation in sufficient detail to undo them or repeat them at a later time. In contrast, the Event generated for this triggering operation would include the time the Trigger was fired; the name (not the actual structure) of the prototype Unit -- Trip Activity -- and the Slot to which the Trigger was attached -- tripDay; the expressions (there may be several) evaluated (abbreviated, if complicated) -- (TestForIncorporateAct) -- and the resulting value; any new Events that occurred during the execution of that function; and the time all the trigger activity completed.

It might seem that this recording would require so much in the way of computational resources as to overwhelm the operation of the system itself. However, because the Event descriptions record high-level information, instead of the full details of the activity, and because most operations have less than ten features to be captured, the amount of information produced is well within reason. Furthermore, the more a system is designed to channel all modifications to system structures through special "access functions", as opposed to having them distributed throughout the code, the easier the addition of Event generating code to the system will be. Certainly it would be easier to build Event generation into the original design of a system, but in any case, little effort is required to add such a facility to an already existing system.. [One of the facilities illustrated in Chapter 9 involved doing this for the MYCIN system. With no previous experience working with MYCIN, it took only a half a day or so to add code to generate Events for the key system operations.]

Event Histories. One goal of the objectification that Event descriptions represent is the recording of a system "history". Although not a new idea, the use of system histories has not been widely explored. One programming environment that includes

something along these lines is INTERLISP. It makes use of one form of system history in its programmer's assistant history list facilities. Information is recorded and maintained about each user input, the result of its evaluation, the side-effects of its evaluation, and other such information, so that subsequently the operation can be looked at, redone (perhaps with different parameters), undone, or altered. [See Teitelman, *INTERLISP Reference Manual*, Section 22, "The Programmer's Assistant".]

Few debugging systems have implemented any kind of history feature. The prominent exception is Balzer's EXDAMS. [Balzer, "EXDAMS -- EXtendable Debugging and Monitoring System," 1969; see also Apple, "The Program Monitor," 1965, for an interesting hardware-level activity recorder oriented towards performance considerations.] That system created a "history tape" during program execution, recording all changes to variable values, choices made at conditional statements, calls to subroutines, iterations, and so on. The idea was that analysis of program activity would be performed *after* and *independent of* the actual execution. With a complete record of all system activity, any desired debugging or monitoring tool could be implemented by combining appropriate routines for searching the history tape with routines for graphical display of the information thereby acquired. The examples given by Balzer include simulating execution -- either forwards or backwards -- at a rate controlled by the user, and the display of the line of code that caused the setting of a particular variable to an indicated value.

The facilities in INTERLISP and in EXDAMS are aimed at the program-level. In keeping with the principles advanced in the present work, a record of the activity of a complex system, such as one based on an AI representation language, must be made in terms of the system's operations and structures. Recording information about each Describe operation in a KRL-1 history would be analogous to recording changes in variable values for a PL/1 program. However, as explained in Chapter 5, the differences between programming languagues and AI representation systems are even more substantial than this. In a traditional programming language, control decisions in deterministic sequences of instructions are made by conditional branches based on simple tests. In contrast, AI system control decisions are based in part on higher-level elements of the context, such as goals and resources. Accordingly, histories of AI system activity must include information about these entities.

The nature of traditional debugging requires the programmer to make an association between particular pieces of execution activity and the statements in the code that had caused the activity. [*cf.* Chapter 4, Section D.] This emphasis conditions the use and implementation of both debugging tools and history facilities. In an AI system, histories should be Event-oriented, with information coalescing around each fundamental system action, rather than around particular parts of the code. The nature

of control in an AI system [*cf.* Chapter 5, Section D.] makes code-oriented traces both impractical and unhelpful. Although some recent work on debugging systems has been aimed at showing the user the connection between activity and code [*cf.* Chapter 4, Section C.], the present proposal rejects that path as an unreasonable direction for monitoring facilities directed at complex systems.

The Structure of Events. The imposition of Event descriptions on the activity of the AI system is intended to result in a detailed representation of what the system has done, along with information as to how, why, when, and so forth. In addition to the production of a history of system activity, such a representation could have significant ramifications for the design of the system itself, a theme to be discussed briefly in the next section. In particular, Events represent a substantial step in the direction of system self-awareness, and they facilitate programming in terms of such concepts as purposes, reasons, resources, and the results of past efforts. For the system to be able to manipulate Events in its own reasoning, the representation of Events describing system activity must be highly structured and handled in a uniform way throughout. Furthermore, the benefits of such meta-organization would be counteracted if this extension increased the complexity of the system and decreased its coherence, as would undoubtedly happen if it simply overlayed the basic implementation. References to Events, whether in user requests issued to the monitoring facility or in system activity itself, greatly benefit from a coherent Event structure.

This direction relates to the "frame" concept currently popular with some Artificial Intelligence researchers, specifically as in the form of KRL-1 Units. [See Bobrow and Winograd, *An Overview of KRL, A Knowledge Representation Language*, Chapter 2.] Not the least of the advantages that can be gained from structuring descriptive information in frame-like ways are those derived from ideas fundamental to property or association lists, from which the frame concept has in part descended. What is important here is the notion that objects of a certain type will have certain typical attributes, and these can be stored as attribute-value pairs. [The first use of attribute value pairs for the representation of semantic information in the context of an AI system appears to have been made by Raphael, in his SIR program. {Raphael, "SIR: A Computer Program for Semantic Information Retrieval," 1968.} He mentions a book *Semantics: The Nature of Words and Their Meanings* by H. R. Walpole {New York: W. W. Norton & Co., 1941.} as one source for the idea that words can be defined by associations, connections, and characteristics. He also points to the description lists of IPL-V {See, for instance, Newell and Tonge, "An Introduction to Information Processing Language V," pp. 364-65 and p. 370.} and the property lists of LISP 1.5 {McCarthy *et. al.*, *LISP 1.5 Programmer's Manual.*} as appropriate implementation structures for the representation of associations.] Not only does this involve a particular *way* of representing information, but it hints at a principle that for all objects of a particular type, there will be a set of standard relevant attributes.

Frame theories are attempts to develop this basic scheme into a robust representational theory. The notion of prototypes arises, along with the distinction between abstract prototypes and concrete individuals. Each prototype makes explicitly available the standard structure for the class of objects that can be described in its terms. The user sees immediately what kinds of information he can expect to find as part of the objects in some class. The resulting association of a name with each piece of information facilitates retrieval processes both within the system and in the programmer's mental efforts. (In the latter case, the value of the association depends on the names being cognitively significant.) The explicit structure provides a vocabulary in terms of which objects can be discussed.

INTERLISP, PASCAL, and other languages have record facilities which are similar in use to property lists or association lists. A difference is that records allow greater freedom in specifying the form of the structure of the representation in record declarations. Although the aspects of objects that are marked as standard by inclusion in their prototype definition are *likely* to be relevant for any individual in the indicated class, their presence for any one individual is not *required* nor is the information for an individual *limited to* the aspects specified in the class definition. Furthermore, an individual may be a member of more than one class, introducing the notion of perspectives. Classes may be collected into hierarchies, with each class inheriting information from its parent classes. [See Minsky, "A Framework for Representing Knowledge"; and Bobrow and Winograd, *An Overview of KRL, A Knowledge Representation Language.*] This avoids the repetition of information shared by several prototypes and can result in greater cognitive efficacy for the user. In this respect, the implementation of hierarchical inheritance serves as a kind of data macro, similar to the program macros of assembler languages.

C. The Uses of Events

There are four general kinds of uses to which the descriptive information contained in Events can be put:

- in concurrent monitoring of system activity;
- for post-execution simulations of the activity
- for interrogation of the history after execution; and
- in the system's own internal reasoning.

The next two chapters will show detailed examples of the first three of these uses. The internal use of Events is outside the scope of this work, but consideration of that theme helps to clarify the nature and value of Event descriptions. Experience has shown that

the more coherently a system is designed from the beginning, and the more it is invested with explicit knowledge about its own activities, the more opportunities there are for "intelligent" decision making in its internal processing.

Concurrent Monitoring. The method of Event generation presented above is meant to result in a flow of information that will tell the user most of what he wants to know about system activity, without burdening him with lower-level details. This is accomplished both through the choice to generate an Event for each system-level basic operation and by making these Events *descriptions* of the activity, rather than a complete record. This Event stream is intended to serve as input to a monitoring facility designed in accordance with the approach presented in the previous chapter. While the AI system executes, the user can watch an organized and visually-oriented display of descriptions of the important system actions. In keeping with the present approach to monitoring, communication about system Events should be much more than simple printouts of their details. The next two chapters will make clear how these Events are *used* in meta-monitoring [*cf.* Chapter 6, Section A, the third part.] rather than simply *shown*.

It would also be appropriate for a monitoring system designed along these lines to give the user some control over what Events, which of their aspects, and how much of their detail he is shown. Depending on the user's immediate needs, he might want to see only a brief sketch of system activity or might prefer to watch a thorough report. Even though the user might choose to restrict the scope of the report he receives during system execution, no change would be made in the number or nature of the Events generated. (The generation of Events is considerably less costly than appropriately displaying them for the user, so when the user calls for less detailed monitoring to make execution proceed more quickly, that goal can be effectively accomplished by filtering the *display* rather than the *generation* of the Events.)

Post-Execution Simulation. In addition to any concurrent displays based on Events, all Events are recorded on a history file. Then, after execution has terminated (or during a break, since there is little difference between the state of things at the "end" of an execution run and the state of things when execution has merely been suspended), this Event history can be used as input to the monitor *without re-execution of the AI system itself*. The generated Events are thus intended to be the sole source of information to be shown to the user. This may not be entirely practical, but it is a simple matter to add any other special-purpose monitoring information and activity to the Event file in the same spirit as the Events themselves. Since the information is equally available *after* execution, system activity can be replayed and monitored independently of system execution. At the end of Chapter 5 it was mentioned that the

slowness and consequent expense of AI system executions made it infeasible to rerun the system several times exclusively for the purpose of acquiring different trace information. Event histories make this kind of replaying a simple and inexpensive process. The user sees just what he would have during actual execution, but without repeating the laborious computations of the AI system.

An important additional flexibility obtained through the use of Event records is that the replay could proceed at a much greater rate than the original execution. The sluggishness of AI systems sometimes makes watching them a painfully frustrating experience. Post-execution replays can proceed at a more comfortable speed. In addition to this inexpensive convenience, there is another important reason why variable-speed replays are valuable. The perception of a pattern depends on psychological processes wherein the pieces of the pattern are observed and in some sense "held together in the mind". For instance, the notes of a musical scale played on some instrument [to use an example from *The Phenomenology of Internal Time Consciousness*, by the philosopher Edmund Husserl {sections 7-16 *passim*.}] are not physically related to each other and do not co-exist in real time [and the same could be said for the frames of a movie]. Yet, the human mind abstracts a pattern from the successive perceptions of individual notes through its abilitity to hold them in Immediate Attention simultaneously. This involves not just the co-presence of the individual perceptions, but the continuous construction of the higher-level, patterned concept: each new note being *added on* to the already existing structure as it is perceived to be a continuation of the pattern. [The notion of the continual adding on of new experiences to present perceptual structures is pursued more generally in another of Husserl's books. See Chapter 8 of *Ideas*.]

What patterns will be perceived from a given set of sensory phenomena depends on the rate at which they are experienced. This is due to the combined effects of the quantitative limitations on human Immediate Attention that were discussed in Chapter 2 and the limitations on how long information remains in Short Term Memory under various conditions. Take as an illustration a story written out on a long scroll. If that scroll were moved through a person's visual field at a normal reading rate, the marks on the scroll would be interpreted as characters of words and the story would be read and understood. If the scroll were drawn by at ten times the normal reading rate, the person would not be able to read words, but might perceive patterns in the length of lines or shapes of paragraphs. If the scroll were moved at a very slow rate, reading would become impossible (beginnings of sentences would be forgotten by the time the final words of the sentence came into view) and the person might focus on the physical shapes of the individual characters.

Thus, there is what might be called a temporal bracketing effect, whereby the rate at which sensory information is received by a person conditions what patterns and meanings will be extracted. The bracket involved is determined by the physical and psychological characteristics of human information processing. Due to the quantitative limits, there is at any given moment a particular subset of the perceptual sequence "together in the mind". The limits on how long cognitive structures remain in Short Term Memory control the boundaries of the subset. Because information processing takes time to accomplish, the rate at which new information enters the system limits the depth to which old information can be analyzed. [It appears that the initial processing of new information normally takes precedence over the continued analysis of old data, so if the STM storage capacity is exceeded, the new information will "push out" the old.] As a result of this temporal bracketing effect, the rate at which information is presented in the kind of monitoring system proposed in this work strongly affects what features of the illustrated activity the user will perceive. Rapid replays will communicate general patterns in the computation; replays that are somewhat faster than concurrent monitoring of the original execution will show the basic outline of the computation; and very slow replays will allow the user to focus in on the details of particular operations. Different speeds would serve different purposes, and the user should be given complete control over the speed of presentation during the replay, not just as an initial setting but as a dynamically alterable parameter.

Interrogation of the Event Space. Not all of the user's needs are appropriately satisfied through dynamic presentations of information describing system activity. The approach to monitoring proposed in the previous chapter emphasizes the importance of meta-monitoring, which enables the user to ask questions *about* system activity. The Event record provides a database for such user inquiries. The availability of facilities for interrogating what could be called the "Event space" would also affect the way in which the more dynamic facilities were used. Knowing that specific questions can be pursued later, the user would be able to focus on the patterns and dynamic characteristics of system activity while watching the execution and replay presentations. Some of the user's needs would be best met by concurrent monitoring displays, some by post-execution replays (at various speeds), and some through static interactive investigation of the Event space.

While the system is executing, the user will want mostly to check that it seems to be behaving reasonably. The details of the computational flow would generally be examined during later replays. Post-execution analysis of specific aspects of system activity could be accomplished by obtaining answers to questions such as "How many Events took longer than 5000 milliseconds of compute time?" or "What activity

resulted from firings of the WhenInstantiated Trigger on the self Slot of the TripActivity Unit?". If the monitoring system and the representation system are suitably integrated, these questions can be phrased directly in the representation language itself and the search performed using the representation system's match and search facilities. (The next two chapters include illustrations of interactive, display-based facilities for this kind of investigation of the Event space.)

Internal Uses of Events. Finally, various advantages would result from incorporating Events into the original design of the representation system, rather than adding them on later solely for monitoring purposes. The system code could create and manipulate Events as a basic mode of operation, instead of dealing with small pieces of *ad hoc* data structures. The infusing of this descriptive meta-representation into the system structure itself endows the system with the ability to answer, for *itself* as well as for the user, questions such as "What objects were affected by Event36?", "What is the goal of the process associated with Event36?", and "What activity contributed to the structure of the currently preferred hypothesis?". To some extent, since such questions will be asked even if there are no Event structures in the system (albeit at the program level), Events simply improve the coherence of the system by providing a language for talking about system activity.

Further advantages can be obtained through internal use of Events in the AI system itself. Events could provide new meta-information not otherwise available to the system, which the system could use in its own internal reasonings. This would be particularly beneficial in the allocation of resources and the scheduling of processes in a multi-process system such as KRL-1. The KRL-1 scheduling mechanisms are generalized, and complex scheduling algorithms could choose the next Process to run on the basis of information obtained from Event descriptions. Choices can be made to run next the Processes meeting descriptions like "the Processes whose activity so far has contributed to the currently preferred hypothesis" or "the Process which has been suspended the fewest number of times". [See Davis, *Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases* for discussion of the use of meta-rules for generalized scheduling in a production rule system.]

Lastly, breakpointing mechanisms could be integrated with the generation of Events, allowing the user to request that the system halt whenever an Event matching a certain description was generated. Again, as for post-execution interrogation, these descriptions could be made in the representation language itself, and the representation system's match or search facilities invoked to compare the description against candidate Events. One benefit of the hierarchical inheritance of information in a representation language such as KRL-1 is that reference can be made to an entire class of objects at

once, including members not yet created. Breaks or traces could thus be associated with entire *sets* of Events through simple, high-level descriptions that apply even to Events to be generated at a time after the request. While it is the previous three uses of Events which are emphasized in this research, the internal use of Events should prove a fruitful realm for investigation.

Chapter 8

Display-Based Tools for Monitoring Facilities

The approach to monitoring the behavior of complex systems presented in the last two chapters is intended to be generally applicable to a wide variety of Artificial Intelligence systems. As shown there, the instantiation of this generalized approach for any particular system must be based on an analysis of the fundamental structures and operations of that system. This chapter describes some specific display-based tools and techniques that proved useful in the implementation of experimental monitoring facilities based on the theory presented in this work. The next chapter shows several examples of facilities that were constructed using the display-based tools described in this chapter in conjunction with the approach presented earlier.

A. A Display-Oriented Programming Environment

Display terminal technology is generally based on either vector drawing or (as is television) raster scanning. To manipulate the display in a vector-based device, the programmer writes a set of instructions in a special purpose language to specify the lines to be drawn on the screen. With a raster-scan device, the programmer sets the values of the bits in a bit map to indicate which points on the screen should be dark and which light. Some of the frustrating and limiting features of machine-level programming surface again at this low level of display programming. Resources may be available at some installations for constructing the few kinds of display-based facilities that would be widely used -- editors, for example -- however painful the task. For other applications, however, the programmer is left on his own to manipulate the bottom-level bits or vectors, although in some situations, general packages for graphics programming are available at a level analogous to symbolic assembly languages.

The programmer cannot afford the time, effort, and frustration involved in dealing directly with the low-level representations of the display, or even with a middle-level graphics packages, anymore than he could implement an AI representation system directly in a machine or assembly language. Consequently, most uses of display devices are highly experimental, and often the more ambitious goals of a project must be abandoned. Any facilities produced despite this barrier tend to be limited in use to

the environment in which they were originally developed. They are also difficult to maintain and improve, due to the low level of their implementation. Programmers have in general not had access to display facilities which could be manipulated at a level corresponding to that of programming languages for the computer itself.

DLISP, A High-Level Display-Based Programming Environment. Recently, at Xerox Palo Alto Research Center (PARC), Warren Teitelman has developed DLISP, a display-based version of INTERLISP. In his view, the project

> . . . continues and extends previous work by . . . [Teitelman] . . . in developing systems which provide the user with various forms of explicit and implicit assistance, and in general cooperate with the user in the development of his programs. The system . . . makes extensive use of a bit map display and pointing device (a mouse) to significantly enrich the user's interactions with the system, and to provide capabilities not possible with terminals that essentially emulate hard copy devices. For example, any text that is displayed on the screen can be pointed at and treated as input, exactly as though it were typed, i.e., the user can say use *this* expression or *that* value, and then simply point. The user views his programming environment through a collection of display windows, each of which corresponds to a different task or context. The user can manipulate the windows, or the contents of a particular window, by a combination of keyboard inputs or pointing operations. The technique of using different windows for different tasks makes it easy for the user to manage several simultaneous tasks and contexts, e.g., defining programs, testing programs, editing, asking the system for assistance, sending and receiving messages, etc. and to switch back and forth between these tasks at his convenience. [Teitelman, *A Display Oriented Programmer's Assistant*, 1977, the abstract.]

Although in this report, Teitelman demonstrates his display system as a user interface to the INTERLISP programming environment, the individual facilities of the system are useful in their own right. These include the manipulation of "windows" ["The idea of a display composed of multiple, overlapping regions called 'windows' is attributable to and an essential part of the Smalltalk programming system designed and implemented by the Learning Research Group at Xerox Research Center." {Teitelman, *A Display Oriented Programmer's Assistant*, p. 1; see Learning Research Group, *Personal Dynamic Media*.}]; the display of different "background" patterns; and the printing of text in various font styles, sizes, and faces. In keeping with the general INTERLISP approach to system facilities, the DLISP mechanisms for manipulating the display are implemented through ordinary INTERLISP data structures and functions, so they can be invoked from the user's INTERLISP code. That the display facilities of DLISP are embedded within a standard high-level programming environment represents a dramatic difference between Teitelman's system and most other display facilities available today.

A major benefit from this design approach is that it makes generally available a well-developed set of facilities for manipulating a display in terms of high-level concepts such as windows, pieces of text, and background patterns, instead of at the primitive level of vector commands or bit maps. Furthermore, the various display operations can be invoked by the user in the same language which he uses for writing his programs. Consequently, the time and effort required to learn and to use these facilities is minimal. Finally, as the experimental display technology at Xerox PARC

becomes publicly available, the INTERLISP community at large could have access to these high-level facilities for display programming. As a result, systems and individual tools constructed in DLISP do not succumb to the problems which undercut most experimental facilities, especially the difficulties of low-level vector or bit map programming and the practical restriction of systems to the particular time and place at which they were developed. Rather than being short-lived, hardly-noticed efforts, experimentation with a system like DLISP can produce results with wide applicability.

Facilities in DLISP. In the course of the work reported on here, a variety of experimental tools and systems were implemented using the facilities for high-level graphics programming provided by DLISP. This section concludes with a short description of the features of DLISP used in these implementations. Teitelman's original report on his system [Teitelman, *A Display Oriented Programmer's Assistant*, 1977.] contains a more extensive discussion of these features. The primary focus there was the display interface to the standard INTERLISP environnment which Teitelman constructed from the basic DLISP mechanisms. (In fact, the layers of DLISP continue downward, since it makes use of a text editing package for INTERLISP written by J. Strother Moore [Moore, *Text Editing Primitives.*] and a graphics communication protocol designed and implemented by Robert Sproull. [Sproull, *Interlisp Display Primitives*; the protocol is based on the ideas described in Sproull and Thomas, "A Network Graphics Protocol".]) As the present description emphasizes the use of DLISP for general graphics programming, it presents somewhat of a different perspective on the system then does Teitelman's report. To make the description easier to understand, a quotation from Teitelman's overview of the system follows:

> The system . . . is implemented on a version of Interlisp running on MAXC, a computer at the Xerox Research Center in Palo Alto. This computer emulates a PDP-10, and runs the Tenex operating system, so that from the standpoint of the user, the system he is using is Interlisp-10. The raster-scan display used by the system described in this paper is maintained by a separate 65K 16 bit word mini-computer. The minicomputer is linked to MAXC through an internal network, and implements a graphics protocol similar to the Network Graphics Protocol [Sproull and Thomas, "A Network Graphics Protocol".], but specialized for text and raster-scan images. . . .
>
> The user communicates with the system using a standard typewriter-like keyboard. In addition, he has available a pointing device commonly called a "mouse" [English, Englebart, and Berman, "Display Selection Techniques for Text Manipulation".] used for pointing at particular locations on the screen. For those unfamiliar with this device, the mouse is a small object (about 3" by 2" by 1") with three buttons on its top. The system gives the user continuous feedback as to where it thinks the mouse is pointing by displaying a cursor on the screen. The user slides the mouse around on his working surface (causing bearings or wheels on the bottom of the mouse to rotate), and the system moves the cursor on the display. The user indicates that the mouse has arrived at some desired location by pressing one of the three buttons on the top of the mouse. The interpretation of the buttons depends on the particular program listening to the mouse. For example, when the mouse is positioned over a piece of text, and one of its buttons pressed, the corresponding text is "selected." . . .
>
> The user interacts with the system either by typing on the keyboard, or by pointing at commands or expressions on the screen, or an asynchronous mixture of the two. . . . [Teitelman, *A Display Oriented Programmer's Assistant*, p. 2.]

The DLISP user makes considerable use of the mouse. [The mouse is a feature of many of the experimental display-based systems at Xerox PARC.] It is used to indicate text to be selected, including the invocation of menu operations. The act of pointing to something on the screen and clicking a mouse button is often referred to as "buttoning" that thing. Another input device that has been experimented with is a keyset with five keys. The meaning of each key or combination of keys can be defined under program control. In addition there are various non-character keys on the keyboard that can be held down while the mouse buttons are clicked; these act to modify or completely change the meaning the buttoning would ordinarily have. Pointing devices facilitate the communication of information from the programmer to the machine. The value of these devices in the context of a display-based system derives from factors in the communication process that relate to those discussed in earlier parts of this work. Pointing requires less time and effort and leaves less room for errors and misunderstandings than expressions in a more symbolic (verbal) mode. [*cf.* Teitelman's similar comment in *A Display Oriented Programmer's Assistant*, p. 2.]

Figure 15 is a picture of how the display screen might appear after a DLISP user has been working for a while. Two other parts of Teitelman's overview that will help in explaining this system follow.

> The user views his environment through a display consisting of several rectangular display "windows". Windows can be, and frequently are, overlapped on the screen. In this case, windows that are "underneath" can be brought up on top and vice versa. The resulting configuration considerably increases the user's effective working space, and also contributes to the illusion that the user is viewing a desk top containing a number of sheets of paper which the user can manipulate in various ways. [Teitelman, *A Display Oriented Programmer's Assistant*, p. 2.]

> One technique heavily employed throughout the system is the use of *menus*. A menu is a type of window that causes a specified operation to be performed when a selection is made in that window. Menus serve a number of important functions. They make it easy for the user to specify an operation without having to type. They act as a prompt for the user by providing him with a repertoire of commands from which to choose. For example, often a user will not remember the name of a command, or may not even be aware of the existence of a command. [Teitelman, *A Display Oriented Programmer's Assistant*, p. 3.]

Menus have been used in many kind of systems over the years. At PARC, menus too are associated with the Smalltalk programming language. [In addition to *Personal Dynamic Media* {Learning Research Group, 1976.}, the *Smalltalk-72 Instruction Manual* {Goldberg and Kay, 1976.} may be of some interest.]

DLISP Entities. In DLISP, windows may look onto documents, desktops, or several other kinds of entities. Documents are sequences of characters; the text may come from a file or may be added through editing operations performed by the user. Desktops are (imaginary) surfaces on which windows reside, simulating the effect of the top of a desk covered with various pieces of paper. A window may be repositioned with respect to the object it views; this is called scrolling. Text inside of a document window may

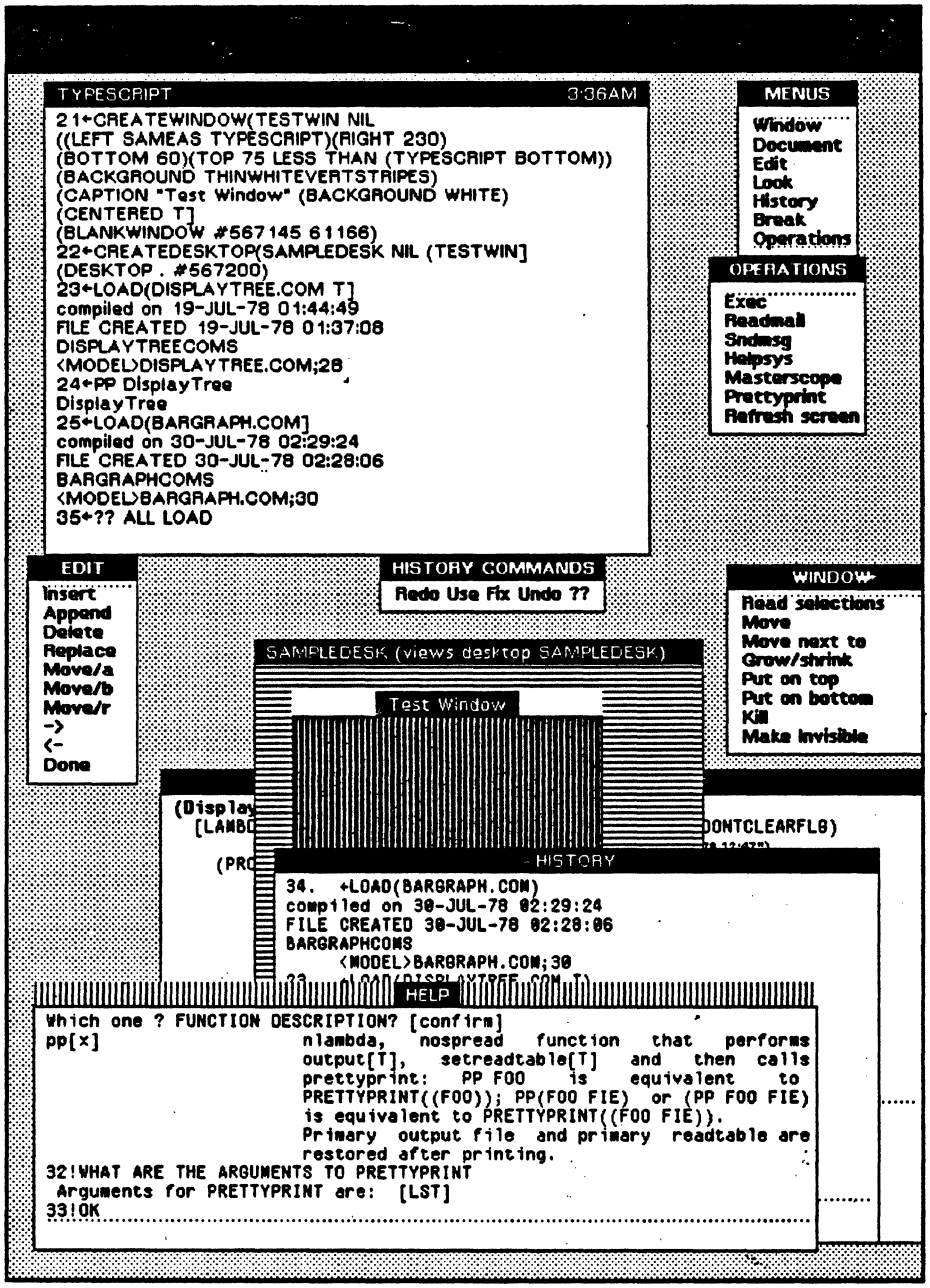


Figure 15. A Typical DLISP Scene

This is how the screen might look during a session with DLISP. Different activities are associated with each window. The usual top-level INTERLISP type-in and responses take place in the TYPESCRIPT WINDOW. HISTORY and HELP are display versions of standard INTERLISP facilities. Underneath those in this example is a window used for prettyprinting INTERLISP structures. The small windows are menus. SAMPLEDESK and Test Window were created by the user. [This and the other pictures like it which follow have been made through a DLISP facility which at the user's request creates a file representing the state of the screen. This "snapshot" can then be printed out on a suitable device.]

be selected. Selected text is indicated by being displayed with its "color" and that of its background reversed. The normal mode of printing is black on white; normally, therefore, selected text is displayed in white on black. Various system facilities make use of selected text. For instance, with only one or two clicks of the mouse button one can cause the text selected in an indicated window to be inserted into the current type-in stream. Graphical objects inside windows can also have actions associated with them which are performed when the user points to them and clicks a mouse button.

Text can be displayed in a variety of type sizes, styles (serif, sans-serif, etc.), and faces (bold, italic, bold italic, or regular). Areas of the screen, such as the parts of windows or of desktops where nothing else is showing and subregions of windows (analagous to the panes of glass in a multi-paned real-world window), are displayed in one of a variety of backgrounds available in the systems (black, white, grey, horizontal stripes, wavy lines, etc.). Lines of a designated thickness can be drawn from any point on the screen to any other, but there are higher-level facilities available in the system that deal with many of the special case line-drawing situations. For example, the user can easily specify operations such as "draw a line four dots wide around the boundary of the pointed-to window", either through a single menu operations selected with the mouse or by adding a simple expression to his program code. Text can be displayed on any background, and in either black or white, as is appropriate for the particular background. Text can also be underlined.

In Figure 15, note that different windows are used for activities associated with each of several INTERLISP facilities, such as those for interrogation of the history list and for questions about system functions, and that the user may create his own windows. Windows which are partially obscured can be brought "on top" simply by buttoning their caption area. DLISP manages the display, in conjunction with a program running on the minicomputer that is part of the system. DLISP determines what parts of windows are obscured by other windows and are therefore not to be displayed. When a window is scrolled, when a window which previously obscured some other windows is made invisible, when a window obscuring other windows is put "on bottom", and at other times when new text must be displayed, only the contents of the altered portion of the screen must be recomputed. The significance of this and other support activity for the utility of DLISP as a general graphics programming environment is that the programmer can manipulate text strings, background patterns, windows, and other high-level entities without having to worry about the details of the display itself.

B. Graphical Tools for Monitoring Facilities

The availability of DLISP provided a unique opportunity for investigation of the concrete advantages of graphical communication, both in general and with respect to the specific requirements of monitoring systems for complex Artificial Intelligence systems. A major motivation of the present work was the conviction that sophisticated display tools could contribute greatly to the power of a monitoring system. This theme was pursued at several levels, through the development and exploration of individual graphical tools, ways of giving different emphasis to individual areas of the screen or pieces of text (using different backgrounds or fonts, for instance), and use of the DLISP facilities to organize flows of multi-facted information (in particular, by sorting information into separate windows). The utility of pointing devices (such as the mouse) and other input media (*e.g.*, the keyset) was also of interest.

The overall experience in using DLISP, both generally and in the context of the experimental monitoring facilities of the present work, demonstrated that a significantly greater flow of information could be understood than would be possible faced with an ordinary terminal or even a graphics terminal serving as a fast teletype. Not only is the communication rate increased using a display-based system, but the information presented using various graphic mechanisms could be substantially more complex than in a printout based purely on characters. In addition, the convenience and the pleasant effects of DLISP contribute to the utility of any monitoring tools and techniques based on that system. The remainder of this section describes some of the individual tools that were implemented in the course of the reasearch reported on here. These use the graphics programming capabilities of DLISP, and are intended to be useful both in the construction of monitoring facilities and for display-oriented programming in general. The tools include mechanisms for displaying tree structures and for creating and manipulating bargraphs. Also described are some ways of utilizing certain basic features of DLISP to organize and clarify the presentation of information. Multiple windows and variety in the appearance of text are of particular importance in this respect.

Tree Displays. In AI systems, process states and data are often tree-structured (or, in some cases, more generally graph-structured). The inherent two-dimensional nature of these structures makes it difficult for the programmer to visualize the interconnections among the nodes based on a linear printout. Because traditional output devices lack facilities for drawing lines and allow only limited control over the positioning of text on a page, they are are not suitable for printing out tree structures in an appropriate form. (However, through the use of indentation and special characters (dashes, vertical bars, etc.), it is possible to produce crude tree-like printouts

on an *ad hoc* basis.) With appropriate graphics facilities, such as those included in DLISP, it is possible to generate clear and pleasing analogical displays of trees. In addition, the interactive nature of the display can be utilized to allow the user to refer to particular nodes or connections by pointing at them. Then, perhaps in conjunction with a menu, a command can be issued to cause some operation on the selected node or connection to be performed. These operations can include requests that the system show further information about the node or connection than would be feasible to display in the tree itself.

The algorithm that was developed for determining where to place the nodes of a tree is fairly complex, since few *a priori* assumptions can be made about the general shape of the tree. Taking into account the dimensions of the space allotted to the tree display, the algorithm assigns tentative locations for the nodes on the assumption that the tree is relatively balanced. (Specifically, the assumption is made initially that all nodes on a particular level have about the same number of successors and that nodes at levels deeper than the shallowest terminal nodes are not themselves the roots of widely-branching subtrees.) Then, overlapping nodes are shifted by a series heuristic procedures, as are the ancestors and successors of those nodes. The tree display is highly parameterized, allowing the user to specify such details as the width of the line drawn around each node, the minimum distance separating nodes, and a function to be called whenever the rectangle representing a node is "buttoned". Figure 16 shows how a list structure representing a simple tree is displayed with this program and, by way of contrast, how it is printed using INTERLISP's "prettyprint" facility.

This tree-displaying program illustrates a theme presented earlier: a facility with wide applicability is made generally available, saving the implementers and users of each particular monitoring system from having to construct it themselves. This is not simply a matter of saving the time and effort of others. Programmers working on AI projects are chronically overburdened. Provided with only a bit-map or vector level display facilities, it would be almost impossible within the constraints typical of such projects for programmers to produce "luxuries" like routines for the graphical display of tree-structures. Even with the kind of high-level facilities for graphics programming available in a system like DLISP, the construction of a tree-displaying mechanism is a substantial task. It is important that such generally useful tools be available *at the start* of efforts toward implementation of a monitoring system, otherwise it is unlikely that they will ever get built. At best, the unavailability of such tools would result in frequent duplication of effort, as each group implemented its own version of the same facility. At worse, desirable and powerful communication tools would simply be left out of these systems.

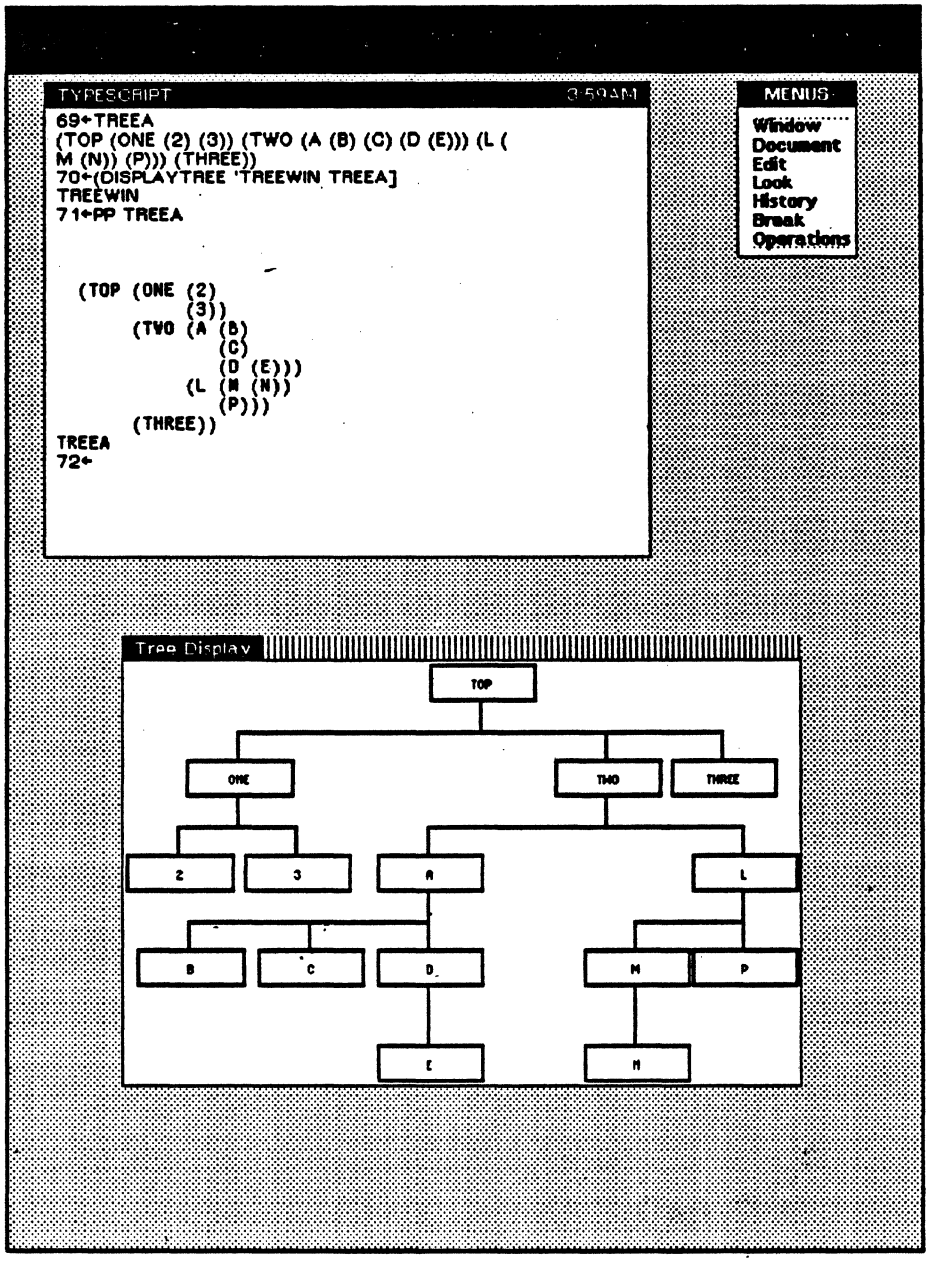


Figure 16. The Display and Printout of a Tree Structure

Trees are easily represented as list structures. The top half of this scene shows how a simple tree represented in this way is printed out by the INTERLISP prettyprint facility. The bottom half is a more analogical display of the same tree, created with a tree displaying package implemented using various basic DLISP structures and operations. The thickness of the lines connecting the nodes, the thickness of the borders of the rectangles, the font in which the node contents are printed, various aspects of the shape of the rectangles, constraints on the separation of the rectangles, and other features of the tree are all parameterized.

Bargraph Displays. The effectiveness of analogic information presented in graphical form was discussed in Section B of Chapter 6. One of the tools constructed in this project was a generalized bargraph package with which the programmer could create and manipulate bargraphs at an appropriate conceptual level. This allows the programmer to refer to the "colors" of each bar, the size of a bar or its pieces, the bar captions, and other aspects of a bargraph without having to worry about the details of the DLISP structures which actually implement those features. The package also replaces certain INTERLISP system facilities with bargraph displays representing the information they print out. One of these is for the STORAGE function. It serves to demonstrate how information which would otherwise require significant time and mental effort to gather and process can be absorbed "at a glance" when presented in an analogical, visually oriented form.

The INTERLISP STORAGE function prints out a table of data types, indicating for each one the number of words of storage allocated to the type, how many of them are in use in the current state of the system, and how many are used by the basic INTERLISP system itself. In INTERLISP, once a piece of storage is assigned to a particular data type it cannot be reallocated to another one. Thus, it is undesirable to allocate much more storage to a type than will ever be required for it. In large INTERLISP systems where free storage is at a premium, it becomes important to carefully tailor the allocation of storage to each data type. The appropriate amounts are determined by invoking the STORAGE function at various times during system execution and analyzing the information it provides. While studying the resulting tables, one often looks for data types which have much more storage assigned to them than is being used. The top half of Figure 17 shows a representative INTERLISP STORAGE printout. The bottom half of the figure contains the bargraph version of the same table. Detecting overallocated data types given the information as presented in the table is a bothersome task, consuming a fair amount of time and effort. By contrast, in the bargraph display, the large areas of black, which denote assigned but unused storage, are noticed painlessly and almost instantly, *at a glance.*

In a traditional programming environment, the progress of an iteration can be traced by printing out messages such as "J = 5", where J is the iterative index. Unfortunately, this information is interspersed among other inputs and outputs, such as queries from the program, user input, and other trace information. The greater the number of processes being traced, the worse this problem becomes. Moreover, the user may not remember where the iteration started or where it is supposed to terminate. In fact, these values may be calculated by the program itself instead of being specified by the user in advance. This latter problem can be solved by including the start and end

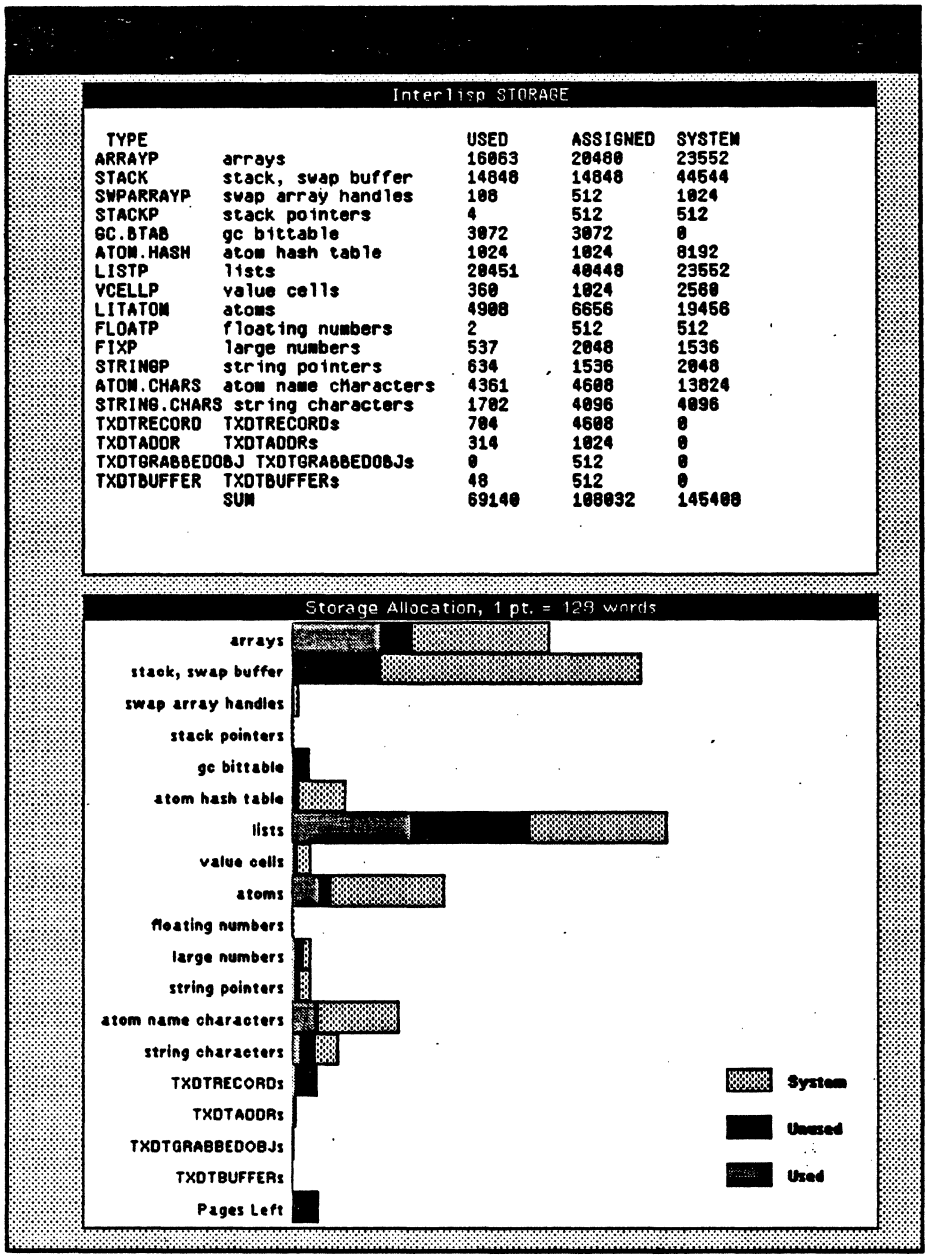


Figure 17. The INTERLISP STORAGE Function and its Display as a Bargraph

The TYPESCRIPT WINDOW in this figure contains a typical INTERLISP STORAGE function printout. The lower half of the scene shows a display version of this function. The display version was created with the generalized bargraph facility implemented using DLISP structures and operations. This illustration demonstrates the ease with which one can find the data types having large amounts of unused storage assigned to them when the information is presented analogically than when it is provided in a table of numbers. For instance, one notices *at a glance* large areas of black, which denote unused storage.

points in the trace printouts. (An appropriate format might be as an ordered triple ‹initial value, current value, termination value ›.) However, even with all the relevant values available, it is not always immediately obvious how far the iteration has progressed. For example, how far along is an iteration whose progress is represented by the triple <-20, 12, 60>? These values represent information at too low a conceptual level; they are *fundamentally irrelevant* to the question at hand, namely "How far has the iteration progressed?". The generally best way to show the extent of progress of an iteration in *symbolic* form would be as a percentage, omitting altogether the initial, current, and termination values.

Although this novel sort of iteration trace would be an improvement over the more traditional methods, it still has the disadvantage of being symbolical when the essential fact being communicated is better expressed *analogically*, for reasons discussed in Chapter 6, Section B. Taking a cue from the charity drive thermometer publicity device mentioned there, the criteria of the present approach to monitoring could be effectively met by representing the progress of an iteration as a partially filled-in bar. Mechanisms for accomplishing this are included in the bargraph package, and Figure 18 shows how such a bar looks on the screen. Several important things should be noted about this device. First, one can see *at a glance* approximately how far along the iteration is. Secondly, the *exact* state of an iteration is rarely of interest, so the approximate nature of the perception is not only easier to comprehend, but is also quite acceptable. Third, and this came as an unexpected bonus, the bargraph can be seen and understood from across the room! This permits one to go off and perform some other activity while waiting for a long iteration to finish, without having to repeatedly interrupt what one is doing to walk over to the terminal, find the appropriate line of information, read what it says, and figure out how much longer the iteration will continue before it terminates. Finally, the information remains in a designated location rather than getting lost in lengthy printouts or "scrolling" off the screen of a display terminals simulating a teletype, and the user knows exactly where to look to find it. [This aspect of the cognitive efficacy of graphical displays will be discussed again in the last part of this section.]

The value of bargraphs as analogical representations can also be appreciated from another perspective. Because we are so much better at the visual perception of relative spatial extent than we are at arithmetical calculations (even simple ones), the visually presented information is much more easily grasped than the equivalent numerical data. Because such a visual presentation ties in to natural, real world perception, its significance is experienced almost *tangibly*, not simply as an abstract conceptualization. Because visual mechanisms perceive relative extents approximately, unnecessary (and, in

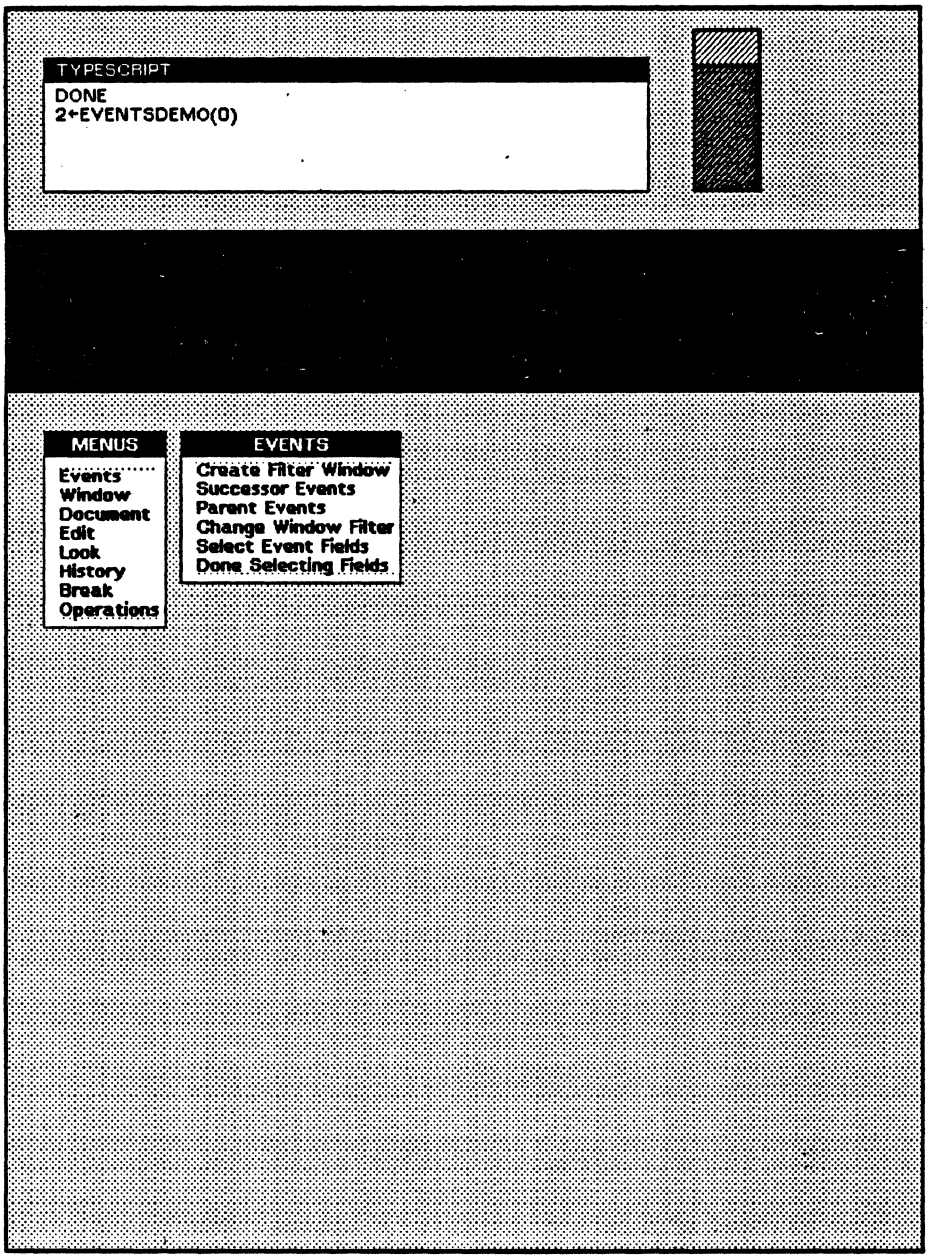


Figure 18. The Use of Bargraphs to Monitor Iterations

When the function EVENTSDEMO is executed (during user interaction with INTERLISP, as shown in the TYPESCRIPT window), an index of Events is created. This process is somewhat time-consuming, and it is useful to be able to see how far the iteration over the set of Events has progressed. The bar in the upper righthand corner is an analogic representation of the progress of that iteration. The lighter area fills in with the darker pattern as the iteration proceeds. With this kind of monitoring tool, one can see *at a glance* the status of the process, and there is no interference with the mainstream user type-ins and program printouts.

fact, distracting) details are filtered out, thus eliminating information and increasing the density of *meaning* in the communication. The simplicity of arithmetical calculations may lead one to erroneously minimize the burden they can be. Even people with much experience in the manipulation of numbers cannot perform the subtraction and division necessary to calculate the percentage of an iteration's progress as easily as they can simply *see* the relationship when presented in a visual, analogic mode. An interesting factor here is that *estimating* the answer to a simple arithmetical problem may be no easier or quicker than finding the *exact* answer. Estimation of an aritmetical result is often accomplished by starting the calculations and stopping when a suitable accuracy has been reached. Mentally setting up the problem and performing only its initial steps is a significant distraction when compared with the ease of hardly-noticed visual perception.

Finally, the most important advantage of bargraphs for representing the progress of an iteration cannot be communicated through the static illustration of Figure 18. By watching the bar grow (or shrink) one grasps immediately the rate of change and intuitively senses approximately how much more time will elapse before the process completes. This second order effects illustrates for a primitive case the third thrust of the approach to monitoring presented in Chapter 6 -- that dynamic processes are best understood through dynamic displays instead of through sequential snapshots of discrete states. Of course an iteration does go through discrete states, as does the bar reflecting its progress, but if the changes to the display are made at sufficiently small intervals (like a motion picture's rapid display of individual frames), a perception of motion occurs. As awkward as estimating relative extent numerically is in comparison to perceiving it visually, estimating the *rate of change* of that extent is even more awkward. In contrast, rate of change in spatial extent seems to be perceived about as directly as the extent itself.

Use of Fonts. The availability of a variety of font sizes, styles, and faces turns out to be quite useful for visually differentiating separate pieces of information that appear in the same locale. The INTERLISP system at Xerox PARC has for some years now been equipped with a multi-font hardcopy output facility, in which function names may be printed in bold face, comments in a small italic font, CLISP keywords in a small sans-serif font in bold and italic face, and so on, depending on the user's tastes. Figure 19 shows part of a page of INTERLISP code as produced by this facility. Such printouts are substantially more readable than is line-printer output, for several reasons. One is the clarification obtained by the use of different font sizes, styles, and faces to distinguish different kinds of information. Another reason is that the use of small fonts for unimportant details (such as comments) improves the continuity of text

```
(RPAQQ TREECOMS ((FNS TREEDEPTH NTREETERMS TREELEVELS MAPTREE ENDORDER)
                 (VARS * TREEVARS)))
(DEFINEQ

(TREEDEPTH
  [LAMBDA (TREE)                                        (* edited: "10-NOV-77 10:22")
    (if TREE::1=NIL
      then 1
      else (for SUBTR in TREE::1 bind TMP MAX←0 do (if TMP←(TREEDEPTH SUBTR) GT MAX
                                                     then MAX←TMP)
            finally (RETURN MAX+1])

(NTREETERMS
  [LAMBDA (TREE)                                        (* edited: "10-NOV-77 10:24")
    (if TREE::1=NIL
      then 1
      else (for NODE in TREE sum (NTREETERMS NODE])

(TREELEVELS
  [LAMBDA (TREE)                                        (* edited: "17-NOV-77 16:39")

          (* takes a tree in normal list-form and returns list of the nodes at each level, -
          i.e. ((T1) (T11 T12 ... T21 ... T1J) (T111 T112 ... T121 ... T211 ... T1JK) ...) -
          where the nodes are of the form (contents child1 child2 ... childN) just as they are in the input TREE)


    (PROG ((NEWTREE (<NIL>))
           NEXTLEV)                                     (* NEWTREE is a TCONC ptr)
          (TCONC NEWTREE <TREE>)
          (while NEXTLEV←(for NOE in NEWTREE:2 join <!- NODE::1>)
              do (TCONC NEWTREE NEXTLEV))
          (RETURN NEWTREE:1])

(MAPTREE
  [LAMBDA (TREE VISITFN MAPTREELEVEL)
    (DECLARE (SPECVARS MAPTREELEVEL))                   (* edited: "17-NOV-77 16:20")
                                                        (* traverses TREE depth first,
                                                        applying VISITFN to each node)
    (if MAPTREELEVEL=NIL
      then MAPTREELEVEL←1)
    (APPLY* VISITFN TREE)
    (for CHILD in TREE::1 do (MAPTREE CHILD VISITFN MAPTREELEVEL+1])

(ENDORDER
  [LAMBDA (TREE VISITFN ENDORDERLEVEL)
    (DECLARE (SPECVARS ENDORDERLEVEL))                  (* edited: "11-NOV-77 13:29")
                                                        (* Traverses a tree in endorder,
                                                        applying VISITFN to each node
                                                        (i.e., after visiting its children))
    (if ENDORDERLEVEL=NIL
      then ENDORDERLEVEL←1)
    (for CHILD in TREE::1 do (ENDORDER CHILD VISITFN ENDORDERLEVEL+1)
                          finally (APPLY* VISITFN TREE])
```

Figure 19. INTERLISP Code Printed with Font Information

This figure shows part of a page of INTERLISP code as produced by a font-oriented device. Function definitions are "chunked", and labelled through a large font and bold face. Insignificant details are printed in small fonts. Text indicating program structure, such as CLISP words are set of in yet another way. Note how much more text can fit in this amount of space then would be possible with an ordinary printer. [Narrower lines were used in this figure than would be in a typical printout, so that the function definitions would fit within the confines of the text borders of this report; this dilutes somewhat the effect of having large amounts of information on a page.]

by allowing more characters to be packed into each line and more lines onto each page without important information being obscured. Similarly, long functions can fit on a single page, greatly improving their readability. Finally, each function is visually set off as a separate "chunk", with its name displayed in a large font and bold face so that each such chunk can be readily identified.

The same kind of advantages can be obtained in a display-based monitoring system. Ways in which different fonts and faces were used in various monitoring facilities will be illustrated in the next chapter. One use of fonts which is common to many of these monitoring tools relates to the Events discussed in the previous chapter. In the course of experimentation, a standard method was developed for displaying the information contained in the Events generated for fundamental AI system operations. The information identifying the Event was displayed in bold-face, descriptive keywords in italics, and details of the Event in a regular face. [Figure 20 shows how such font face information is used in the display of a simple Event.]

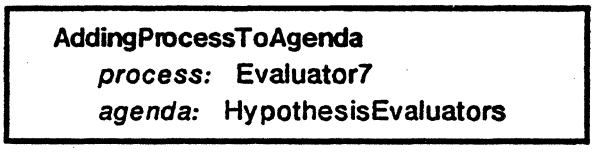


Figure 20. The Display of a Simple KRL-1 Event Using Different Type Faces

This particular way of displaying the details of an Event was found to be so generally useful, especially in conjunction with sorting information into various windows (discussed below), that a function was constructed which could be invoked by the user of the facility to display other structures in this way. This gives the user a ready-made tool, much like that of the tree-displaying function, which he can employ for his own purposes. The designers of the INTERLISP system made a point of giving the user access to system functions that seemed generally useful, and the experiences of INTERLISP users have proved the value of this attitude. During experimentation with various monitoring tools and facilities, care was taken to notice functions which, although originally constructed for internal use, might be more widely applicable. When it seemed worthwhile, the original functions were generalized, paramaterized, or extended, in keeping with this policy.

Organization with Windows. The use of separate windows to organize text and graphical displays provides several cognitive benefits. Trace information issuing from a complicated program is likely to include messages concerning several different processes. In a linear teletype printout, the different streams of information are indiscriminately intermingled on a line by line basis. Poring over such printouts

looking for the lines relevant to one particular entity or question can be a tedious undertaking. Windows are an effective way of disentangling the confusion by assigning each stream of related information to its own window. Each window would then show a unified flow of information all of which related to some particular process or to the development of some particular data structure. Some information might conceptually belong to more than one trace stream and could easily be added to several windows at once, as in the case of information concerning the progress of some process through which several alternative hypotheses had been developed. In addition to the clarity this kind of organization establishes for the display, the user knows where on the screen he will find the information he wants, and he doesn't have to waste his time and effort searching for it. [This is valuable for analogical displays also, as mentioned in the discussion of bargraphs above.]

Another advantage of window-based displays derives from their being "scrollable". Information which moves up past the top of the window as new information appears below can be brought back by user button operations or under program control, in a straightforward way. Thus it is necessary to retain in the visible part of the display only the most recent information, since the old information can always be regained. One advantage of this is that there is space on the screen for the simultaneous presence of information from a greater number of sources than would be possible if old information had to be retained as long as it might be useful, lest it be permanently lost. Finally, the user can review the stream of information collected into a window by starting at an earlier part and scrolling through all the subsequent information (rapidly, if desired).

Overlapping of windows, along with the organization of sets of them into desktops, facilitates maintenance of the screen display. The information visible is that which was most recently displayed or used, with windows less recently active partially or completely obscured. Typically, when information is printed to a window in DLISP, the system redisplays that window "on top" of all the others, so that it is visible in its entirety. As long as any part of a window remains unobscured, the window can be brought back "on top" -- to full visibility -- through appropriate button operations. Even if a window is completely obscured, it can be uncovered by moving the upper windows to the bottom using the mouse buttons, or by typing in appropriate LISP expressions.

Further, as windows are left unused in DLISP, they may "fade away" -- their contents are partially obscured by the superposition of a sort of mottled pattern. If a window continues to be inactive, this effect is increased in strength; finally, the window is made invisible. This aids in clearing unused windows from the screen (they can

always be redisplayed -- although they are not shown, their contents remain in the same state as while the windows were visible). This mechanism also helps to indicate to the user, without destroying the contents of the window, that the window has been unused for a period of time. Another way of accomplishing similar effects was investigated during the experimentation. In this method, as a window remains inactive it is reduced in size, so that it remains visible while taking up less area on the screen. Eventually, the window would shrink to a mere icon of itself, but still remain visible. In this way, the user would still see *what* information was available, and where, but would not be distracted by the details of inactive parts of the display. Although not developed fully, this proved to be a useful alternative to the standard DLISP mechanism for indicating inactive windows.

C. Independent Monitoring

There are a variety of relationships that can exist between a computational system and a debugging facility for that system, in the context of both traditional and Artificial Intelligence programming. The simplest facilities are the ones which the user writes for himself and are called directly from within the code. These usually involve the printing of messages to the terminal about the stage the computation has reached or the values of some important variables. Some of the standardly available debugging tools are also invoked from the user's code, through statements such as TRACE ON and TRACE OFF. Such tools are applicable mostly to compiler-based languages. In an interactive system, some trace facilities may be invoked from outside the code, as with INTERLISP's BREAK package. Here, the code is modified by one of the functions that causes tracing or breaking, as called by the user.

Some debugging tools have no direct connection with any source language system. One group of these comprises machine-level debugging tools such as DDT. Low-level orientation is not the only way to achieve this generality, however. EXDAMS, for instance, was designed to be a source-level facility which could be applied to different languages. [Balzer, "EXDAMS -- EXtendable Debugging and Monitoring System," 1969.] Most of the recent attempts at display-based debugging systems envision the display system as an integral part of the programming environment, not as a separate system. [*e.g.*, Yarwood, *Toward Program Illustration*, 1977; and, to an extent, Swinehart, *COPILOT*, 1974.] The experiments performed for the present work involved the construction of monitoring facilities which, although tailored for some particular AI representation language, were not built within those systems. This approach, which will be referred to as **independent monitoring**, arose initially out of necessity, but it proved to have several important advantages.

Because of inherent limits on the total amount of storage available in INTERLISP, it is impossible to load a system as large as KRL-1, the system for which most of the monitoring experiments were performed, into DLISP. It was therefore necessary to implement and execute the monitoring facilities independently of the representation system. For post-execution investigation of the recorded history, this of course presents no problems. The monitoring system is run and the history recorded, after which the monitoring system is started up and the history analyzed. The same is true for post-execution replays of system activity. However, if the monitor is to run concurrently with the representation system, information must be exchanged between the two. With this in mind, various mechanisms for providing a communication channel between the two systems were investigated.

File-Based Communication. The simplest technique experimented with involved the use of files which were written and read by both the AI system and its monitor, separate "jobs" running on the same computer. This an approach which also has been used by the PSI group at the Stanford Artificial Intelligence Laboratory in its automatic programming project. [See Green, "The Design of the PSI Program Synthesis System" for a brief description of this project. The use of files to communicate among portions of the system does not seem to have been explicitly documented anywhere, but has been discussed in various talks presented by Green and the other members of the PSI group.] The file-based approach can be implemented fairly easily, but is very inefficient, as file-oriented input and output is quite slow compared to in-core computation. This technique involves frequent examination of the file state by each system to see if anything new has been written. Using files for inter-system communication thus tends to overburden both systems with a lot of slow input and output activity.

Communication through Page Mapping. Another technique which was explored utilized the page mapping facility available in the TENEX operating sytem. Page mapping refers to a mechanism whereby a page of storage in one file or a page of memory in one TENEX "job" is "mapped" -- made to be effectively the same as -- a page in another file or job. This is accomplished through the use of the TENEX PMAP JSYS. [See the *TENEX JSYS Manual*, Section 2, pp. 55-58. The TENEX operating system provides various support facilities, many involving input/output operations. These are invoked through calls to the monitor using the JSYS instruction in the PDP-10 assembler language.] Whenever a change is made to one of a pair of mapped pages, the same change occurs in the other one. Using the PMAP JSYS, pages can be mapped between two INTERLISP systems running as forks of the same job (or two separate INTERLISP jobs, each mapping pages to a file shared between the two jobs). This mapping can serve as a communication link, since whatever is stored by either job into a mapped page is directly and immediately available to the other job. There are limitations, however, in that INTERLISP pointers,

strings, and other non-simple structures cannot be shared, for a variety of technical reasons [especially those having to do with garbage collection, which may result in data being moved and pointers being changed]. In practice, only binary-coded numbers and characters can be exchanged using mapped pages.

An experimental semaphore system for interjob communication was constructed using page mapping. Special locations were reserved for flags which indicated that messages were waiting or that they had been processed, along with information as to where the messages were located. The messages were phrased in what amounted to a special-purpose code for the tasks at hand. This method of communicating information between the two jobs served its purpose, but was awkward to implement and inflexible in use. However, for systems which require several jobs to share information, this holds promise as a feasible approach. The technique is efficient, and could well be the best way to proceed where the intricacy of the multi-job organization would justify the effort required to develop the semaphore protocol and communication language to the extent necessary for effective performance.

Network Communication. Eventually, [with considerable assistance from Larry Masinter, at Xerox PARC] a general INTERLISP package was put together that provided a mechanism for establishing connections between two INTERLISP jobs running on either of the two MAXC computers at Xerox PARC. This package used the Xerox Ethernet [See Metcalfe and Boggs, *Ethernet.*] to send streams of characters between the two jobs. The continued development of this facility resulted in various extensions, including the ability to establish the same kind of connection between INTERLISP jobs running on computers at different sites on the ARPANET. [The ARPANET is a network sponsored by the Advanced Research Projects Agency of the Defense Department which interconnects many computer installations in the United States and a few in Europe. Many of the ARPANET sites are involved in Artificial Intelligence Research. The network has been in use for quite a few years. Despite earlier hopes for more elaborate utilization of the facility, it has served mainly for the exchange of messages and files among users at the various sites and to allow a user at one site to log in to a computer at another site.]

The INTERLISP network communications package makes use of mechanisms which already existed within the TENEX operating system for establishing and using network connections. These mechanisms manage the network connections and handle the individual "packets" of information sent over them, assembling these into a continuous stream of characters. This means that the INTERLISP code does not deal with the processes of transmission, receiving, synchronization, and packet assembly through which network communication operates. The package is concerned only with the initial establishment of the network connection. After a connection has been set up with this package, the ordinary INTERLISP input and output facilities can be used to print to or

read from the connection. From the point of view of programs using this facility, the network connection is just another file. [Although one with some special properties. When an "end of file" indicator is encountered during a file input operation, an error condition is raised. In contrast, input operations on network connections will wait until the characters necessary for their completion are received, permitting the system to ask for and await information. Also, network connections have no file-pointers associated with them; input and output operations are strictly sequential, so information communicated in this way is not randomly accessible. These characteristics of network connections are helpful for some purposes, and a hinderance and for others.]

Advantages of Independent Monitoring. In the network-based communication scheme, the monitoring and representation systems are independent, both physically and conceptually. Even if it were possible to embed the monitoring facilities in the object system itself, there are many reasons to keep the two separate. The bulk of the code in the monitoring system is devoted to handling the communication with the object system, organizing the flow of information, and handling the display. Displaying trees, positioning text appropriately, and processing user requests are examples of activities that must be performed regardless of the details of the specific system being monitored, while the amount of system-specific code and data that must be included in the monitor is small. Because the representation systems are themselves general (KRL-1, a production-rule engine, etc.) user applications would require few, if any, special additions to the display facilities. In addition, it would be inefficient to include display management facilities within each monitor. These factors make the separation of the system and its monitor a natural and practical approach.

In addition to these general considerations, various specific implementation advantages are obtained by separating the monitoring system from the representation system. The two systems can be implemented in different programming languages, depending on what seemed appropriate in each case. Then, as was mentioned above, the two systems could run on different machines. This parallel processing utilizes the computing power of two machines instead of one, an obvious benefit. The independence of the two systems contributes to the overall flexibility of each and of their interaction. One system can send off some information or issue a request to the other, then continue its computation. This contrasts with implementations in which the monitoring facilities are part of the programming system. In the more common situation, each segment of the system has to complete its current operation before any other segments can proceed with theirs. Another advantage of this approach is that the ability to run a system at another ARPANET site, with the monitoring system running locally, increases opportunities for experimentation. One of the examples discussed in the next chapter involved the execution of the MYCIN system on the SUMEX-AIM computer while the monitoring system ran on one of the MAXC machines at Xerox

PARC. For a variety of technical reasons, this proved to be much simpler than attempting to transplant the MYCIN system to Xerox PARC.

The separation of the monitoring facility from the AI system facilitates the development of each, since the two efforts can proceed independently. The implementers of one do not have to track program-level changes in the other. The separation of the two systems also permits construction of a suitable monitoring system with minimal perturbations to the representation system. All that needs to be included in the latter is code for communicating to the monitoring system information about fundamental operations, as they occur. If the AI system is structured in terms of Events, as proposed above, this addition becomes a trivial matter. Even without such structuring, determining the key points in the code and adding appropriate Event generating instructions has proven to be a straightforward procedure where tried.

Thus, the two systems can be linked together without the builders of each requiring extensive knowledge about the implementation of the other. The experiments performed for the present work, especially the one involving MYCIN, would simply not have been feasible had it been necessary to weave together the monitoring tools and the representation system into a new hybrid system. To learn the details of an unfamiliar representation system simply for experimental purposes would not have been worth the necessary effort. Finally, the utility of a tool beyond the environment in which it was initially implemented is directly dependent on the ease with which it can be transported elsewhere. An already implemented system requiring only minor additions and modifications for application to a different environment is certainly easier to transplant than code which must be reworked for another system by someone familiar with the details of both.

Chapter 9

Some Experimental Monitoring Facilities

A substantial portion of the research reported on in this work involved experimentation with actual monitoring tools and facilities. The first ones constructed were based on preliminary versions of the approach to monitoring put forth in Chapter 6 and the method of describing system activity presented in Chapter 7. The approach and method were further developed and refined as experimentation revealed appropriate paths to pursue. The tools described in the previous chapter were designed as support components of the monitoring systems. They too were extended and generalized over the course of the experimentation. This chapter illustrates the performance of some of the monitoring facilities which were produced during the later stages of this work.

The investigatory stages of the research involved predominantly the development of an extensive display-based environment for KRL-1. In addition to monitoring tools, this facility incorporated applications of DLISP text-editing capabilities to the problematic task of creating and altering KRL-1 structures, including those found in user code. [The implementation of KRL-1 data structures, because of their complexity and the influence of efficiency considerations, makes it virtually impossible to edit those structures using the INTERLISP editor. Therefore, the KRL-1 user must edit these structures by printing them to a file, altering them with a text editor, then reading them back in. To those accustomed to the editing facilities provided by INTERLISP, this represents a throwback to more primitive conditions. One of the major aims here was to provide a text-oriented editing facility that would behave as if an internal part of the KRL-1 system.] The goal was to create a complete display-based interface that mediated and facilitated all interactions between the user and the KRL-1. Each of the basic KRL-1 activities was assigned its own window, and these in turn were grouped into coherent desktops so that they could be moved, made visible, or made invisible together. This was found to facilitate context-switching, providing a form of assistance of considerable value in a multi-faceted environment. [*cf.* Teitelman, *A Display Oriented Programmer's Assistant*, p. 3.]

The monitoring aspects of this display-based environment focused on process and scheduling activity in KRL-1. None of the current KRL-1 application systems make use of the language's process-oriented facilities.' Consequently, although the work on

dynamic monitoring of KRL-1 had a major influence on the theory and tools presented in this report, it is not possible to demonstrate that phase of the research at this time. Nevertheless, some of the tools and techniques developed in the course of experimentation with the KRL-1 dynamic monitoring facilities were found to have applicability in other contexts, both for different types of monitoring and for other representation systems. Thus, some of the features of the experimental KRL-1 system will be illustrated in the two examples which are included in this chapter.

Section A demonstrates a facility that assists the user in his investigation of an Event space created during the execution of a KRL-1 application system. The example chosen for the demonstration was a prototype trip-planning system built by the Knowledge Based Assistance Project at Xerox PARC [At that time, this consisted of Richard Fikes, Bill Laaser, and Warren Teitelman.]. It proved quite a simple task to add Event generation code to KRL-1 and produce a record of Events during the execution of the trip-planning system. The trip-planner uses KRL-1 Triggers as its main control device. Although it is a simple matter to dynamically monitor Trigger activity with the KRL-1 monitoring facilities mentioned above, doing so does not result in an interesting demonstration of their capabilities. This is because the process activity for which the dynamic facilities were designed is highly organized, whereas the purpose of triggering mechanisms is to enable a system to respond to activity at arbitrary times. Since in a trigger-based system there is essentially a linear, if disorganized, flow of processing activity, dynamic monitoring does not contribute substantially to the user's understanding of the system's behavior. What *is* useful is investigation into the characteristics of the whole body of system activity, after the execution has been terminated. The discussion in Section A incorporates illustrations from a session during which the Event space from a run of the trip-planner was analyzed.

The second example in this chapter shows the concurrent, dynamic monitoring of a representation system's activity. MYCIN, an experimental medical diagnosis program developed at Stanford University [Shortliffe, *MYCIN*], was chosen for several reasons. For one, it has been well-publicized. Secondly, it is publicly available as a running system. Also, it exists on the SUMEX-AIM computer; this presented a good chance to demonstrate network-based communication between an independent montor and the AI representation system for which it was designed. Further, MYCIN's knowledge is embodied in a set of production rules, and its processing is therefore very different from KRL-1. This made it possible to test the claim made in the present work that the approach to monitoring and the particular tools developed are not limited to use with KRL-1 but are applicable to AI representation systems generally. Finally, MYCIN is sufficiently well-designed that only a day or so was required to understand the main

points of its implementation and add Event generating code for its fundamental operations. Although the processing in MYCIN is not as complex as that envisioned for a system using the KRL-1 process and scheduling facilities, it is a reasonable substitute subject for demonstrations of the value of concurrent, dynamic monitoring. Section B illustrates the operation of the MYCIN monitor as applied to the processing of one of the stored patient histories which are also available on the SUMEX-AIM system.

A. Viewing the Event Space of a KRL-1 Application System

The Knowledge Based Assistance Project at Xerox PARC developed an experimental trip-planning system, implemented in KRL-1. The system makes particularly heavy use of Triggers, one of the major control mechanisms in KRL. Triggers provide a means for attaching procedural information to declarative structures. The Triggers get fired at appropriate times, under the conditions designated in their specification. A trigger-oriented flow of control is useful when many individual pieces of information enter the system at unpredictable times and in an unpredictable order. The extensive use of triggering thus contrasts with process-based control. The latter requires much better knowledge, on the part of both the programmer and the system, regarding the order and interdependencies of the activity to be performed. Although flexibility can be obtained through the use of elaborate scheduling mechanisms in KRL-1, this flexibility is controlled by the system. The *system* decides what to do next, based on its analysis of current goals, resources, and available information. An unpredictable flow of information hinders those kinds of decisions, since important information will be missing much of the time. Of course, Processes can check periodically to see if what they require is available, and go dormant again if not, but the system will bog down if this occurs too frequently. Under such circumstances, it is often preferable to allow the data to drive the system, through the use of triggers.

Precisely because of this kind of flexibility, a trigger-based system produces a diffuse flow of activity that is difficult to follow and understand. The code which specifies the computation to be performed is distributed throughout the system, rather than collected together in the traditional linear program. The tests that determine whether a KRL-1 Trigger is to be fired are internal to the sytem and so are not readily observable or understandable by the user. Since each Trigger can cause more than one action to be taken, an implicitly tree-structured nesting of activity occurs. Unlike the activity resulting from process-oriented activity, triggered activity does not manifest strong global patterns. As a result, the dynamic monitoring of a trigger-based system is not as important to the user as the post-execution investigation into what happened.

A stabilized early version of the KRL-1 trip-planning system provided an excellent opportunity for experimenting with Events-based post-execution monitoring facilities. Event generating code was added at appropriate places in KRL-1, and a record of Events was collected for a run of the trip-planning system. The remainder of the present section demonstrates a short session with the monitor. It shows the kinds of operations the user can perform, the type of information he can obtain, and the manner in which it is presented. The basic notion is that of a "filter window" through which the user looks onto the Event space. [A conversation with Peter Deutsch at Xerox PARC helped crystallize an early muddled conception of this use of windows into its present form.] The user tells the system that he wants to create a window, and then specifies its location and a filter expression. The system scans the record of Events and collects those which match the filter expression. These are then displayed in the designated window. By selecting which Events are included and which of their aspects are shown, the user can easily construct displays which answer a wide variety of questions.

During this illustrative session, only simple filter expressions were specified. However, expressions such as the following emerged during experimentation and proved to be natural ways of posing various kinds of fundamental questions:

```
(EventType EVENT) = 'TriggerEvent and EVENT:TriggerEvent.type = 'WhenFilled
```

to request all Events describing activity resulting from the firing of a WhenFilled type of Trigger; or

```
EVENT:startTime GT 12300 and 49400 GT EVENT:endTime and not
        ((EventType EVENT) = 'TriggerEvent and
          EVENT:TriggerEvent.protoname = 'PlanTripStep)
```

to ask for all Events that began before $t = 12300$ and ended before $t = 49400$, but not if the Event describes the firing of a Trigger in the PlanTripStep prototype Unit. These expressions are written in CLISP, a standard part of the INTERLISP system. [See Teitelman, "CLISP -- Conversational LISP" and *INTERLISP Reference Manual*, Section 23, "CLISP -- Conversational LISP.] Ideally, these expressions should be in a suitable representation language, such as KRL-1 itself. Unfortunately, implementation considerations made this impractical for the purpose of this experiment. Instead, Events are organized simply as INTERLISP record structures, and can be accessed as such. In addition, some special functions were provided for accessing aspects of Events not directly obtainable through a record access expression [the general form of which is OBJECT:RECORDNAME.FIELDNAME].

(In many of these illustrations, one window has a caption with a background of vertical stripes. This designates the CURRENT WINDOW, a standard DLISP entity which is used as an implicit argument for many operations. The CURRENT WINDOW is the one in which activity has most recently taken place.)

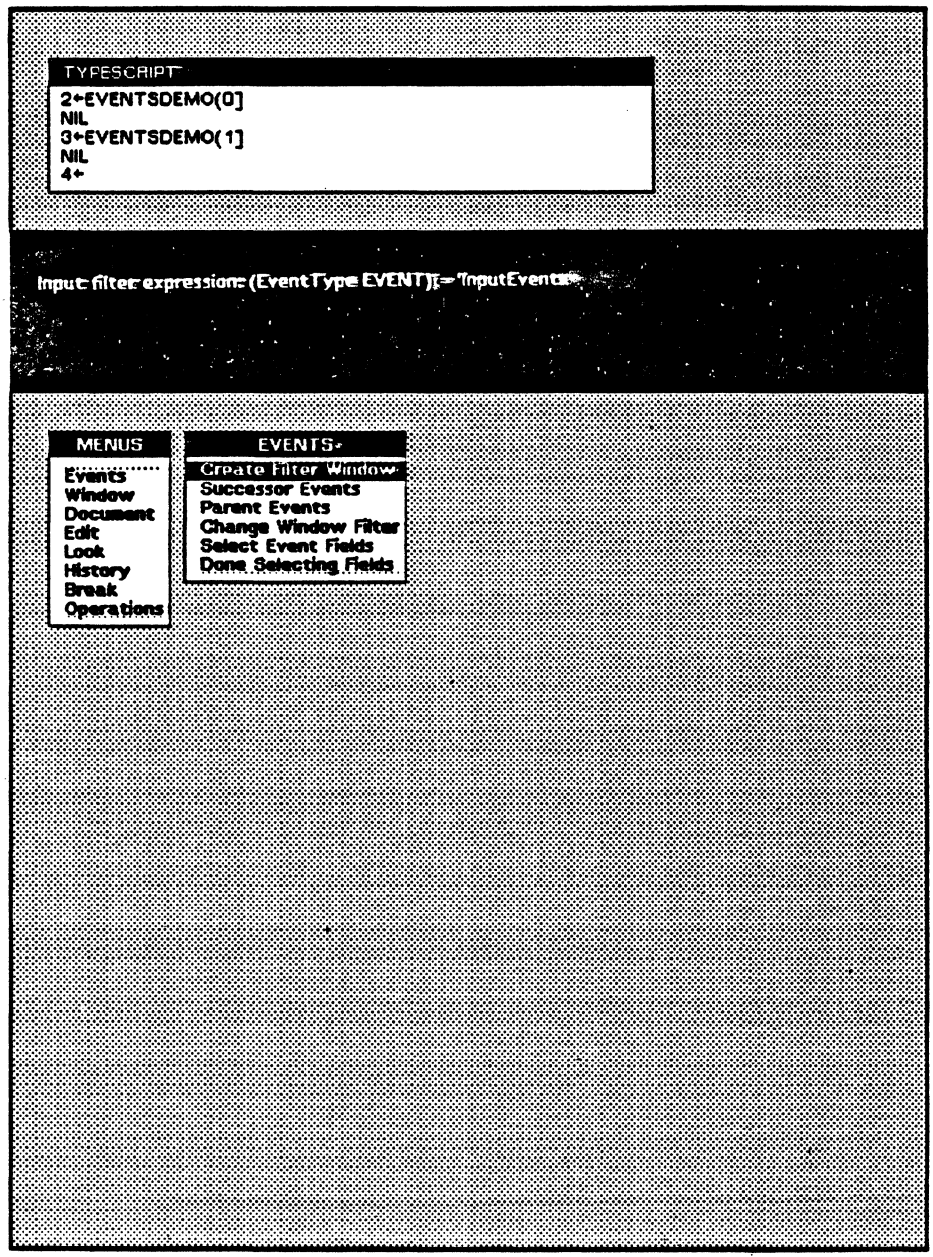


Figure 21. A Representative Interrogation of the Event Space, Stage 1

EVENTSDEMO was called to start the demonstration system. The mouse has been used to select the Create Filter Window operation on the EVENTS MENU. The black area towards the top of the screen is the DLISP PROMPTWINDOW, somewhat enlarged and repositioned. The system has asked the user to type in a filter expression, and the user has responded. [Ideally, these expressions should be in a suitable representation language, but implementation considerations made this impractical for the purposes of the experiments. However, the ordinary CLISP record facilities and several special Event functions (such as the function EventType seen in this example) are available for accessing aspects of Events.]

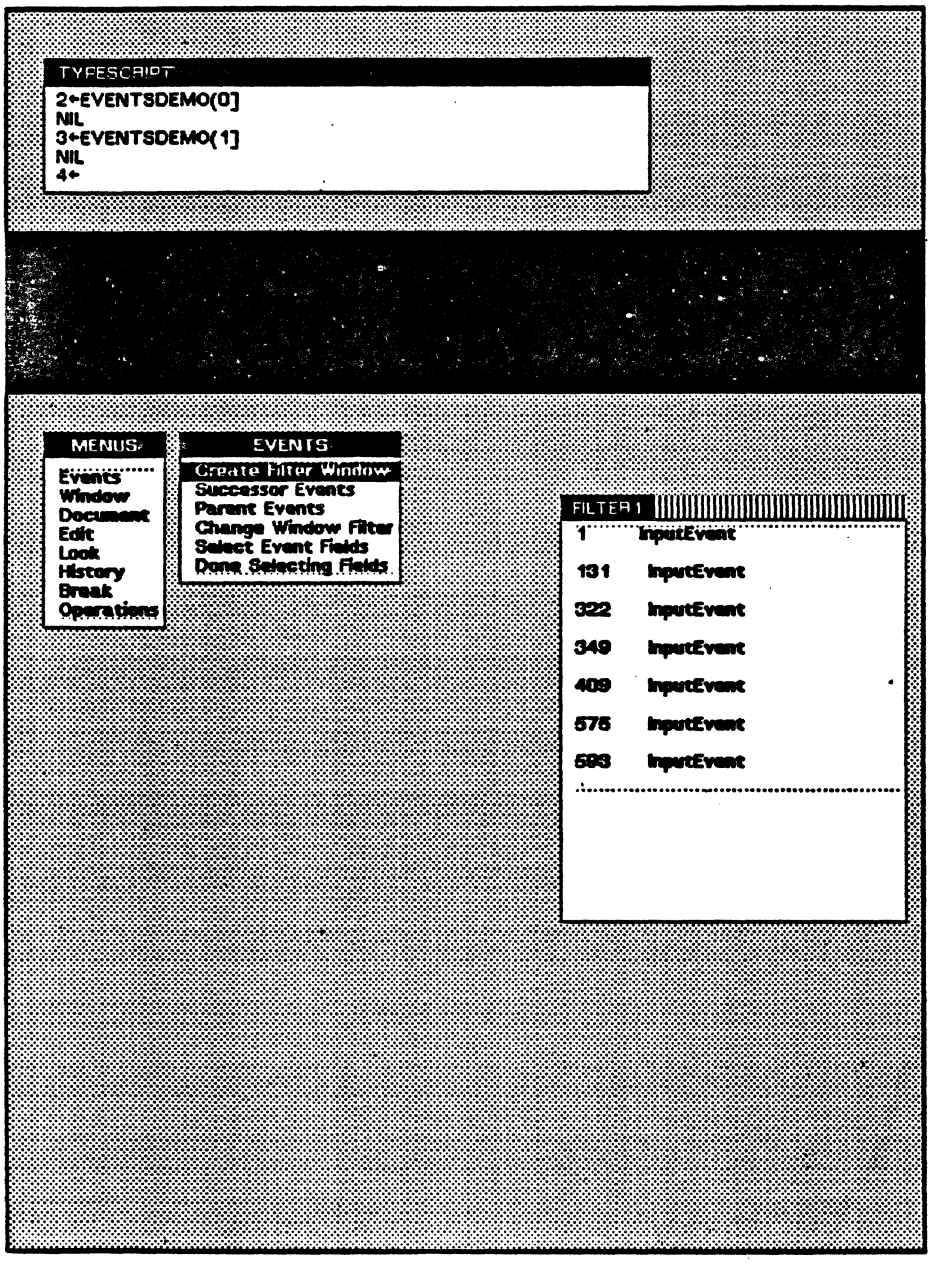


Figure 22. A Representative Interrogation of the Event Space, Stage 2

The system has requested (in a PROMPTWINDOW message) that the location of the new filter window be specified. Two points on the screen were buttoned, to indicate the position of two opposite corners of the window. The window has been created and displayed. The Events record has been searched for all Events matching the filter expression. [A bargraph trace had automatically been invoked to monitor the progress of this search process (*cf.* Figure 18).] Finally, the number and type of each selected Event are displayed (in bold face) in the window. The dots at the top and bottom of the window indicate that the entire contents of the window are visible.

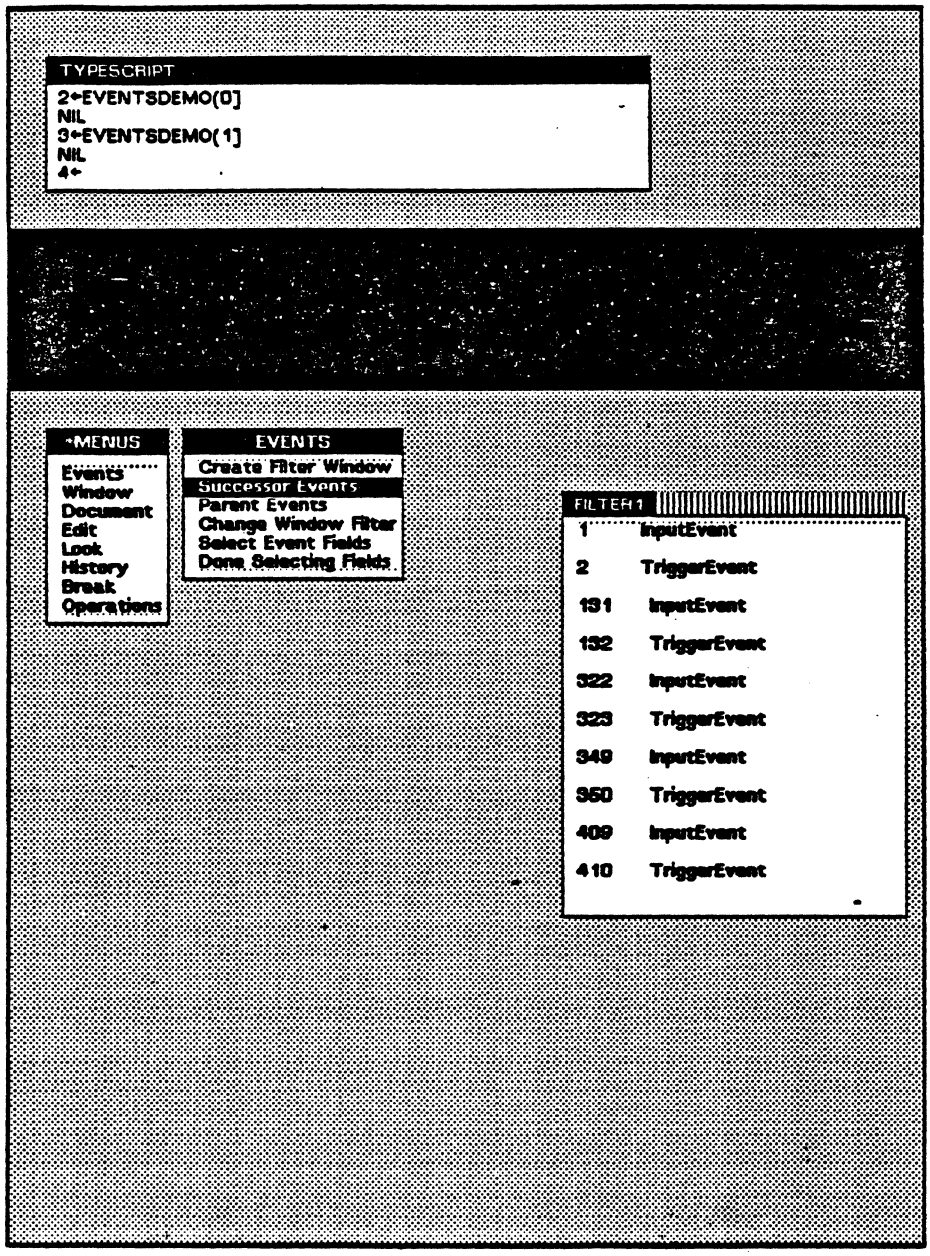


Figure 23. A Representative Interrogation of the Event Space, Stage 3

The Successor Events operation of the EVENTS MENU has been buttoned. This operation adds to a designated filter window all Events which describe activity that was the direct result of activity described by any Event already included in the window. A more complicated filter expression could have been specified initially to include these additional Events. However, the menu operation provides a simple way to accomplish this kind of extension of the set of selected Events. Because of the nature of Trigger-caused activity, this kind of extension is one which the user will frequently desire.

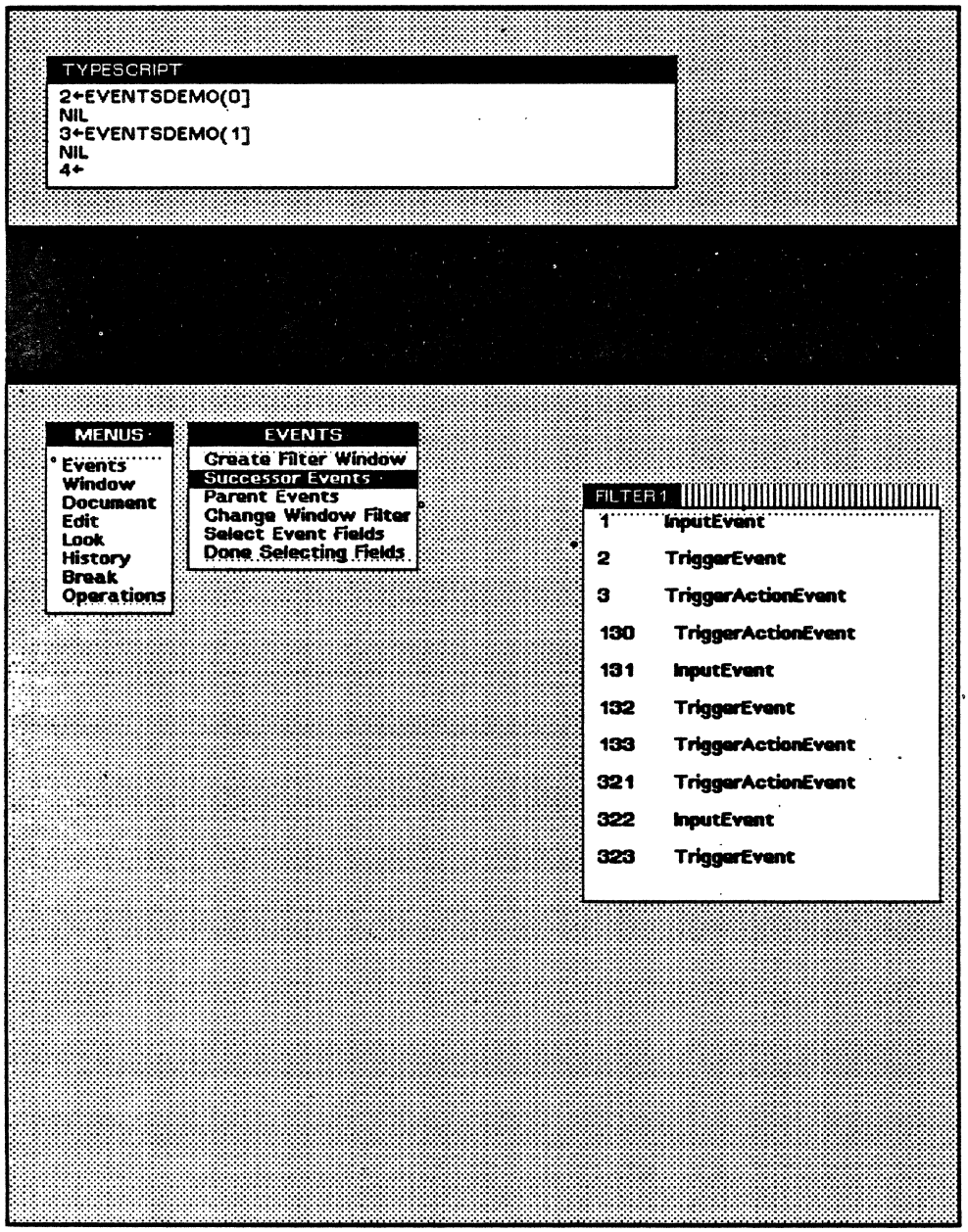


Figure 24. A Representative Interrogation of the Event Space, Stage 4

The Successor Events operation has been repeated. A TriggerActionEvent is created for the evaluation of each form in a fired Trigger (there may be several). Thus, in the Events visible in this snapshot, the activity described by Event 1, an InputEvent, caused a Trigger to fire. Event 2, a TriggerEvent, describes the activity associated with that Trigger firing, including the evaluation of two forms. The latter are described by TriggerActionEvents 3 and 130. As there is no dotted line at the bottom of the window, there is more text below that which is visible in the window. This text can be viewed by scrolling the window contents.

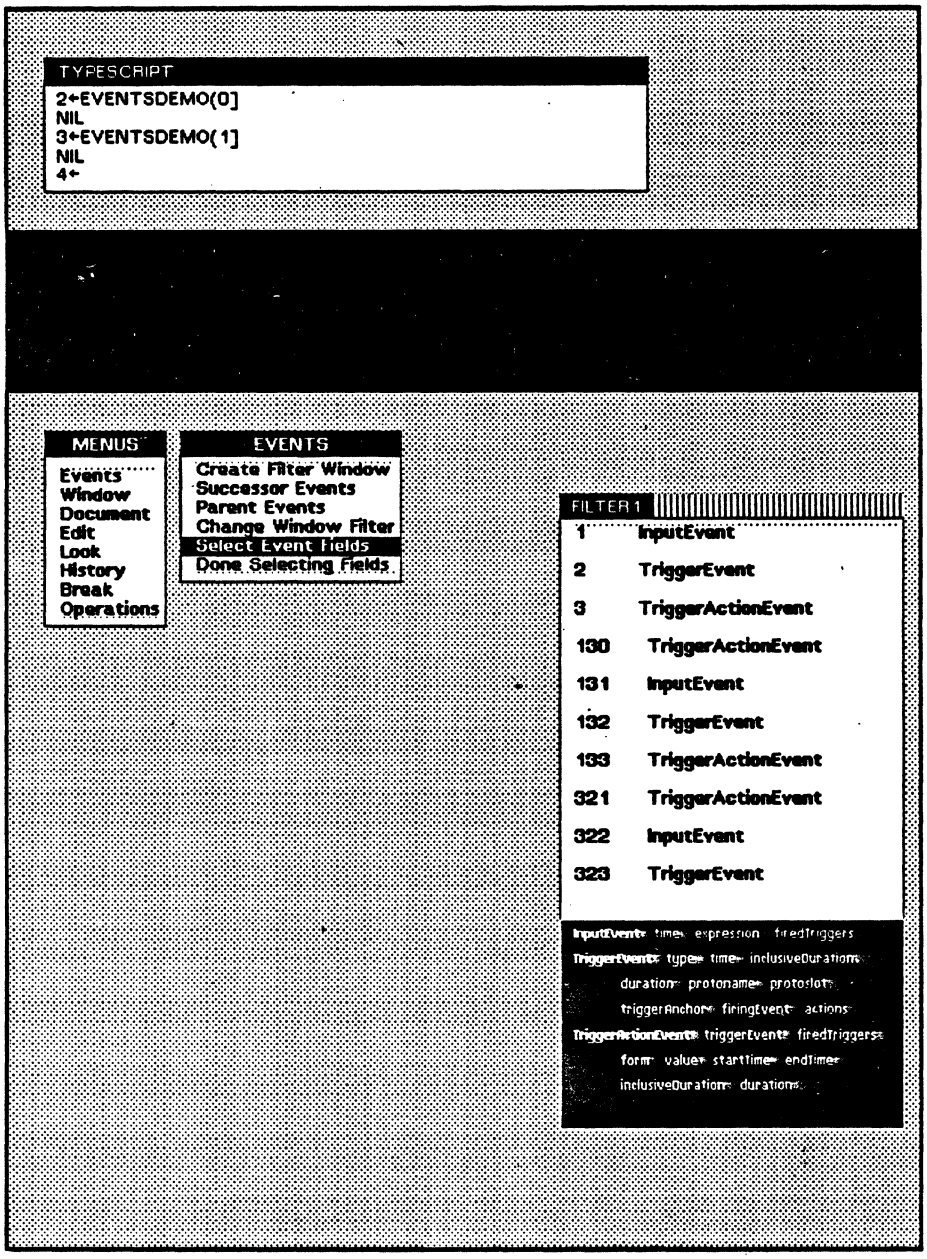


Figure 25. A Representative Interrogation of the Event Space, Stage 5

In keeping with the approach and method described in Chapters 6 and 7, the details are of Events are not shown unless the user asks for them. The Select Event Fields operation causes an Event Field Selector window to be displayed under the designated filter window. The Event Field Selector shows in bold face the types of Events present in the filter window, in this case InputEvent, TriggerEvent, and TriggerActionEvent. For each of these, the names of the aspects that constitute Events of that type are shown in the indented portion of the text.

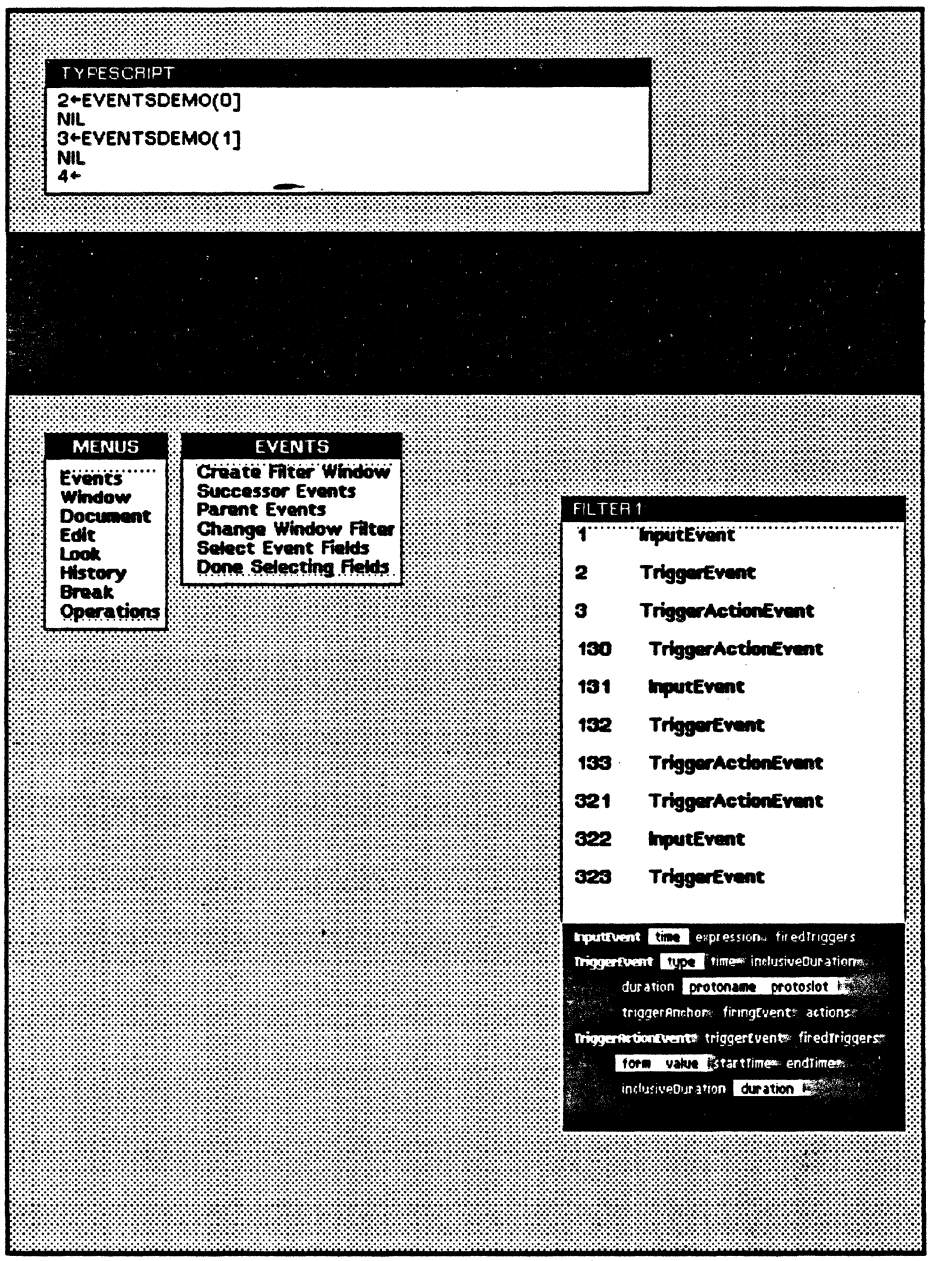


Figure 26. A Representative Interrogation of the Event Space, Stage 6

The user selects which aspects of the Events he wants to see by pointing at field names in the Event Field Selector and clicking a mouse button. The names of the selected fields are displayed with background reversed.

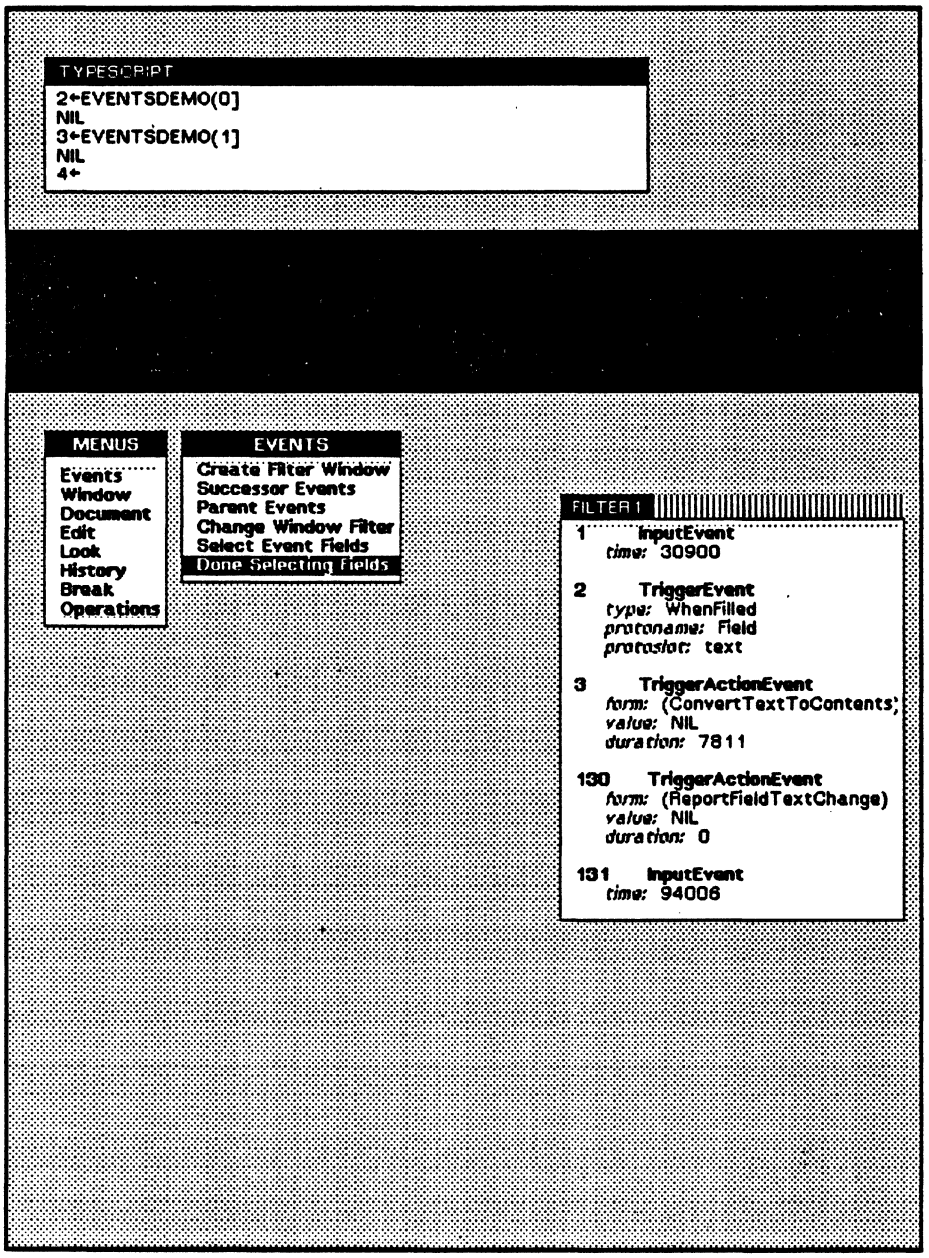


Figure 27. A Representative Interrogation of the Event Space, Stage 7

Some fields were selected, then the Done Selecting Fields operation of the EVENTS MENU was buttoned. The Event Field Selector disappears, and the Events in FILTER1 are redisplayed showing the designated aspects. The name of the field is printed with an italic face to set it off from the information following it. At any time later, the Event Field Selector can be recalled; it will appear with the names of the previously selected fields already selected (*i.e.*, with background reversed).

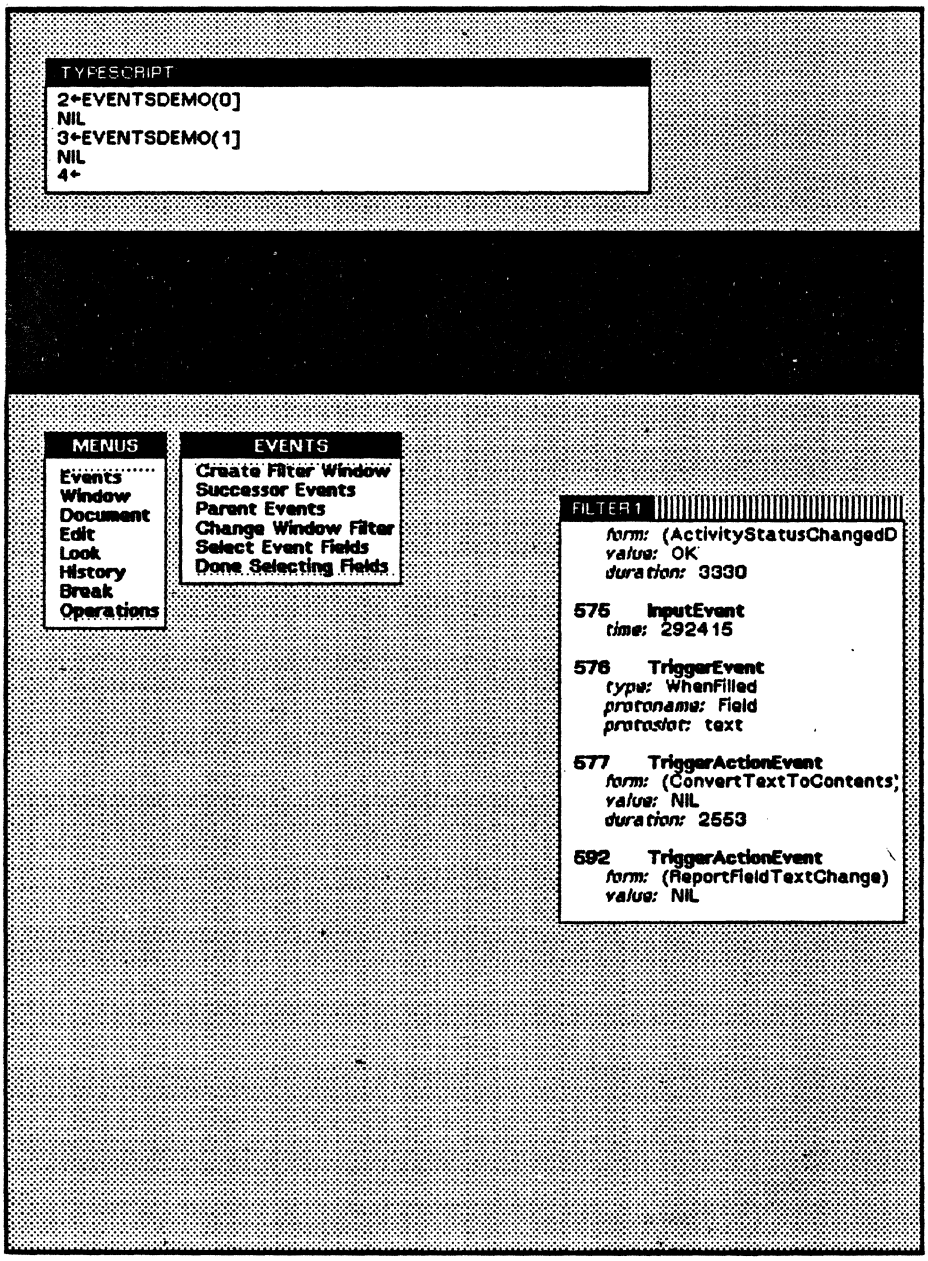


Figure 28. A Representative Interrogation of the Event Space, Stage 8

This snapshot was taken after the window contents were scrolled to view Events toward the end of the set contained in the filter window. Durations are in terms of milliseconds of compute time. The user can choose between "duration", which does not include any time spent performing the activity of successor Events, or "inclusiveDuration", which does include successor Event activity. The "forms" of TriggerActionEvents are arbitrary INTERLISP forms to be evaluated when the Trigger is fired.

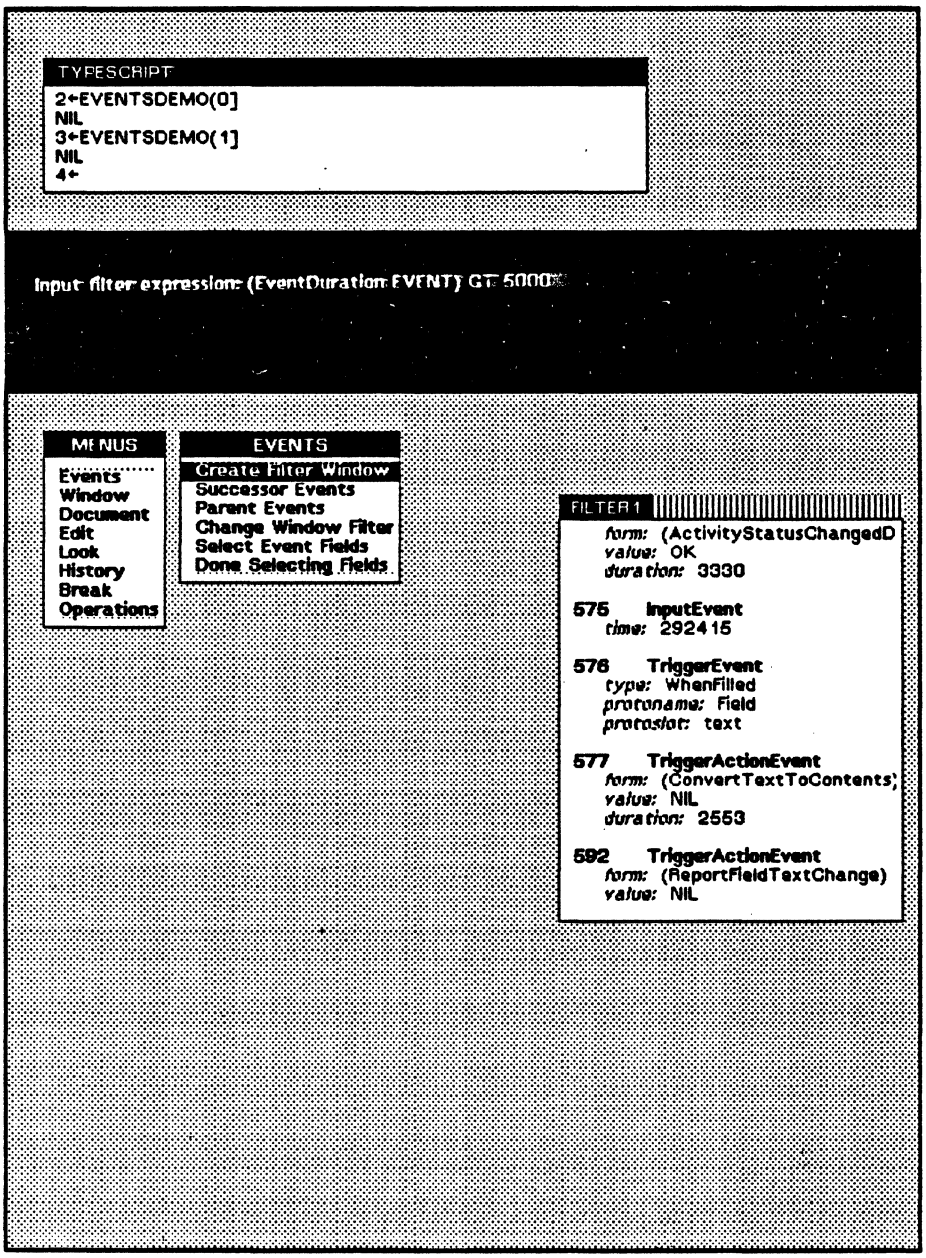


Figure 29. A Representative Interrogation of the Event Space, Stage 9

The user has asked that another filter window be created. This one is to contain all Events which describe activity that took more than 5000 milliseconds to execute (not including the activity of successor Events). The ability to ask this kind of question is a valuable one, and it illustrates the theme of meta-monitoring discussed earlier regarding the importance of answering questions *about* system activity rather than just *showing* it.

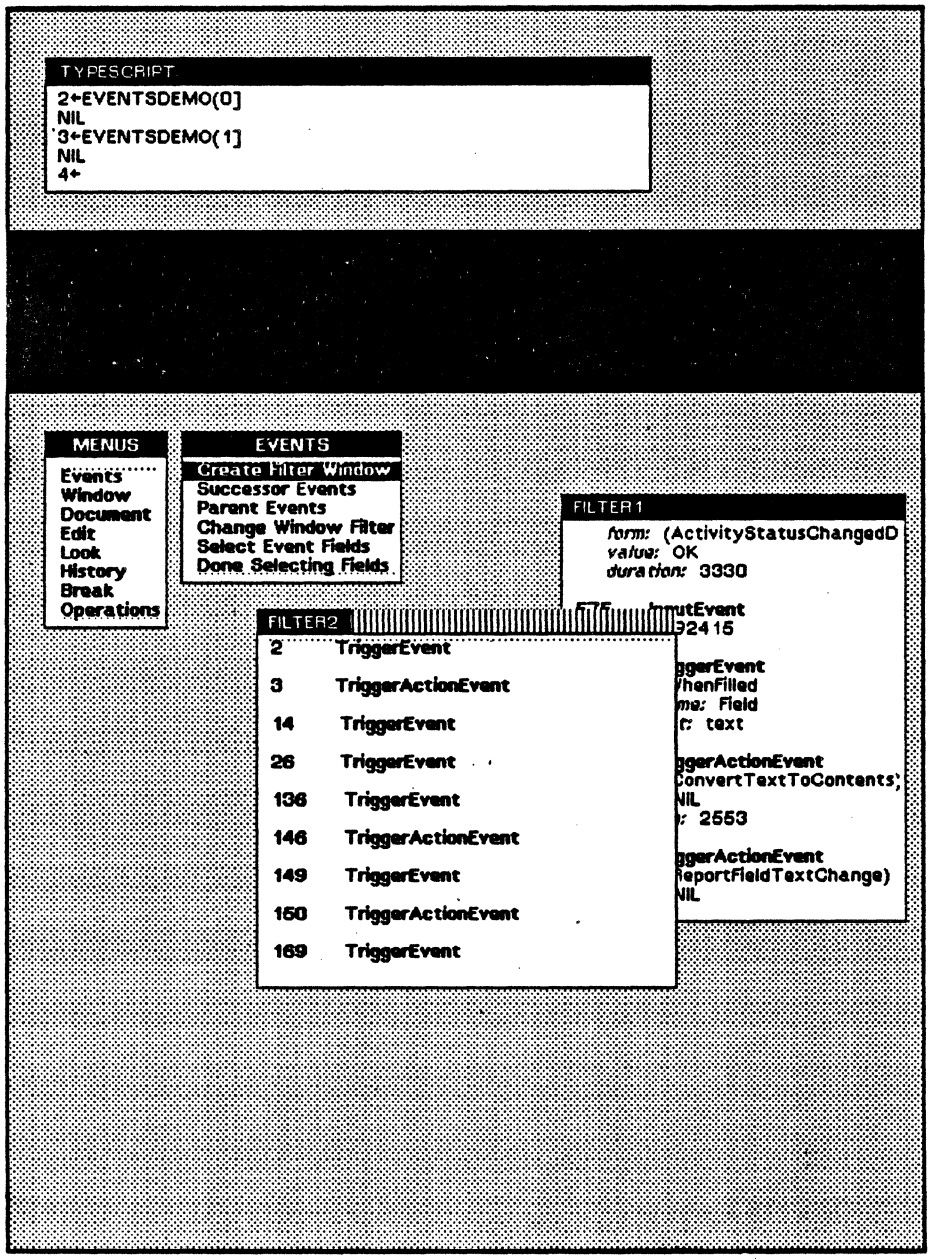


Figure 30. A Representative Interrogation of the Event Space, Stage 10

The user indicated the corners of the new window. The system created and displayed the new window, searched the Event record again, and displayed the selected Events. Note the overlapping of windows. The user has complete control over the dimensions and location of windows. Not only does he specify these when the windows are created, but operations on the standard DLISP WINDOW MENU permit moving windows, changing their shape, and so on. The user can also button the caption of a partially obscured window to bring it "on top" or to put a top-level window underneath the others. FILTER2 is now the CURRENT WINDOW.

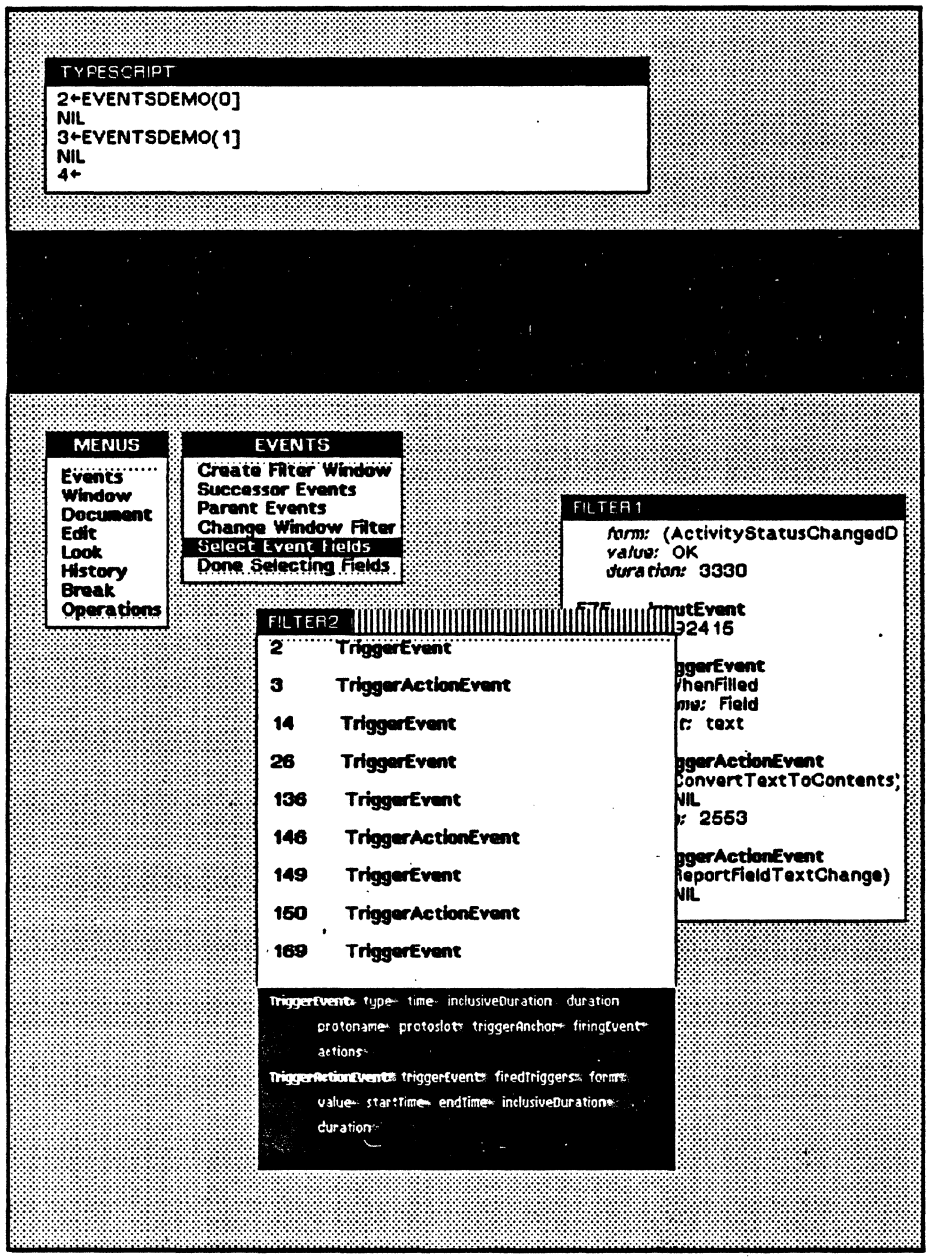


Figure 31. A Representative Interrogation of the Event Space, Stage 11

An Event Field Selector has been requested for the new filter window. Note that as there are no InputEvents included in FILTER2 [The activity described by InputEvents involves little computation that isn't performed by the firing of Triggers.], the names of the fields for that type of Event are not included in the Event Field Selector.

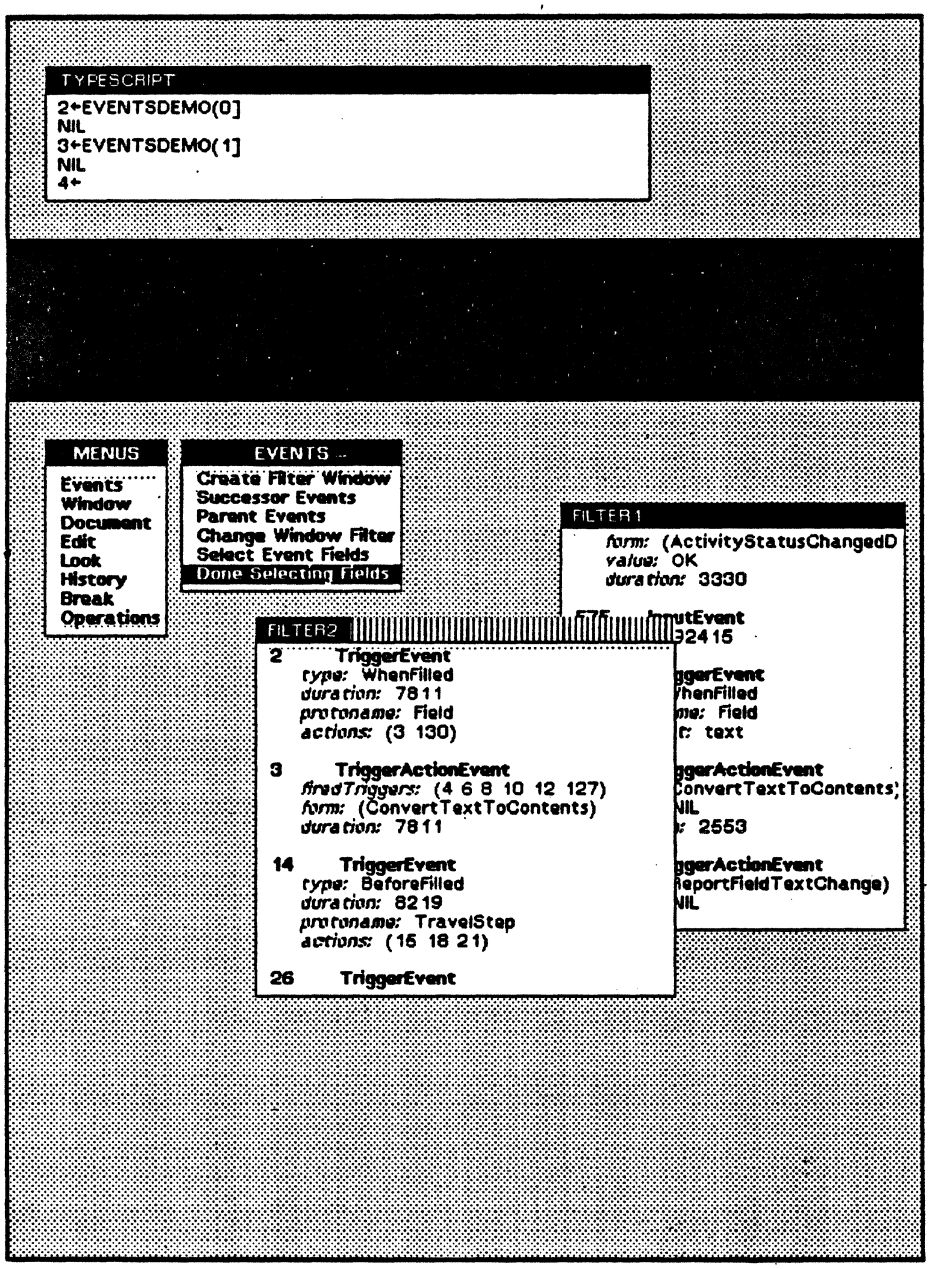


Figure 32. A Representative Interrogation of the Event Space, Stage 12

The user has buttoned Done Selecting Fields, and the selected aspects of FILTER2's Events have been displayed. [The middle button had been used, so the command applies to the CURRENT WINDOW. If the left button had been used, the system would have prompted the user to specify a window by pointing to it with the mouse and clicking the left button.] Notice that the activity described by Event 3 resulted in six different Triggers firing; the numbers displayed are the numbers of the Events representing that activity. The "actions" of a TriggerEvent such as Event 14 are TriggerActionEvents; these represent the activity of each of the forms in the Trigger specification.

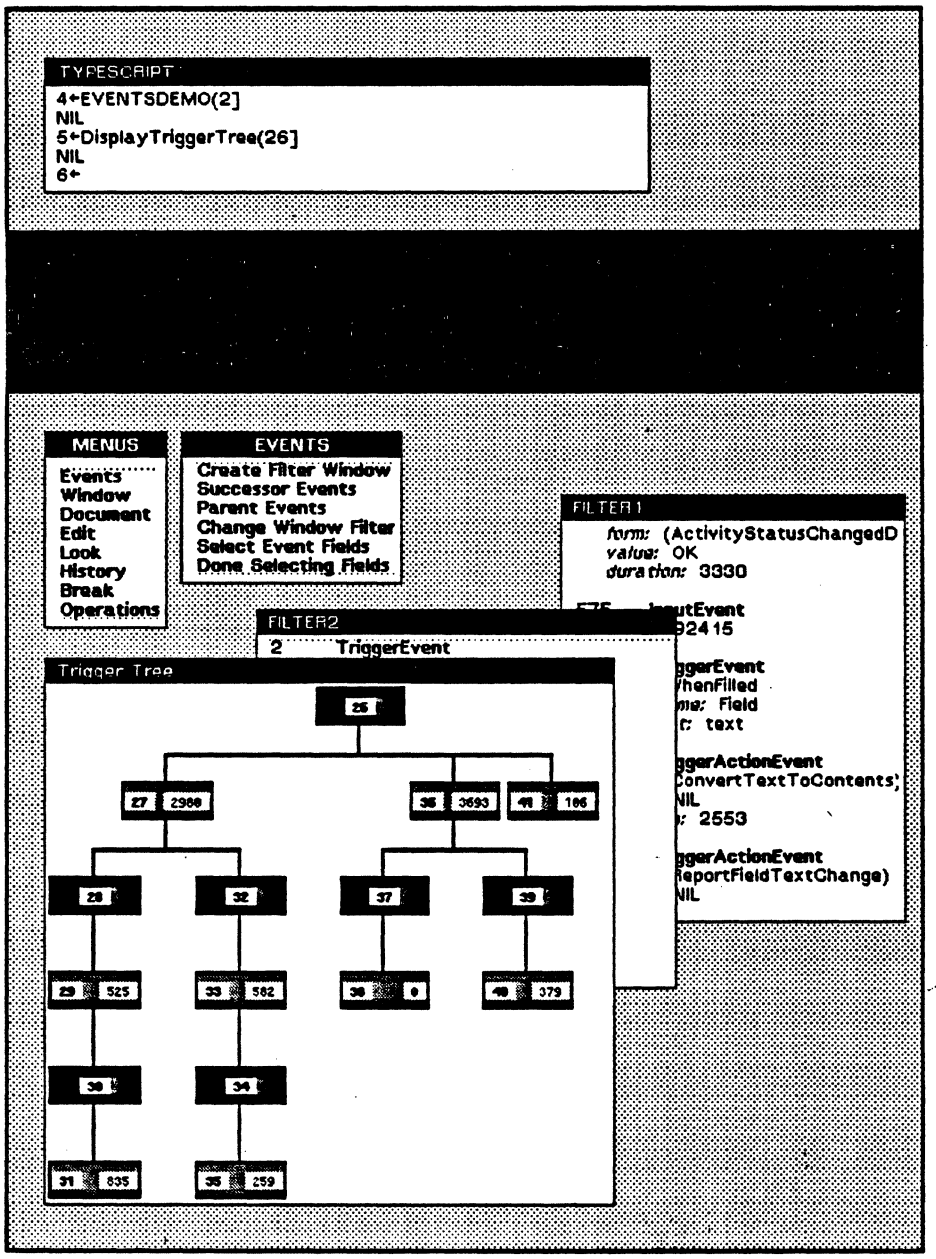


Figure 33. A Representative Interrogation of the Event Space, Stage 13

Finally, the user has typed in a call to the function DisplayTriggerTree, in the TYPESCRIPT WINDOW. This function displays a tree of the Events which are successors to the one specified. The nodes at each level alternatingly represent TriggerEvents (displayed with black backgrounds) and TriggerActionEvents (with grey backgrounds). The (non-inclusive) durations of TriggerActionEvents are shown in the right hand side of the node. [Little activity is performed during the firing of a Trigger other than the evaluation of its forms, so durations are not included for TriggerEvents.] Such displays effectively communicate the structure of system activity.

B. Dynamic Monitoring of System Activity in MYCIN

The approach to monitoring and the scheme for producing descriptions presented in previous chapters were intended to be applicable to representation systems generally. Most of the experimentation took place within the context of various monitoring facilities for KRL-1. To test the generality of the approach, a monitoring facility was was constructed for MYCIN, a medical diagnosis program which uses knowledge stored as production rules. [Shortliffe, *MYCIN*.] In this context, MYCIN was run on the SUMEX-AIM computer, through an ARPANET teletype connection. The development of the monitoring facility proved to be quite a simple task, demonstrating the general applicability of this work's approach. This section presents snapshots of a sample MYCIN execution using the monitor. [The run uses one of the stored patient histories available at SUMEX, but the responses to the questions could just have well have been typed in from the monitoring system -- interactions in the Mycin Typescript Window are treated as a terminal on the SUMEX-AIM system, through the teletype ARPANET connection.]

In the MYCIN monitor, interaction with MYCIN appears in the Mycin Typescript Window of the monitor. A MYCIN option was selected for this demonstration run which caused MYCIN to print out trace information during execution. The MYCIN trace facilities are quite good for a teletype-oriented system. These therefore provide a good basis for comparing traditional traces with the kind of dynamic monitoring activity shown in the lower half of these scenes. Unfortunately, the dynamic quality of the monitor display cannot possibly be captured in a series of snapshots, as has already been pointed out above.

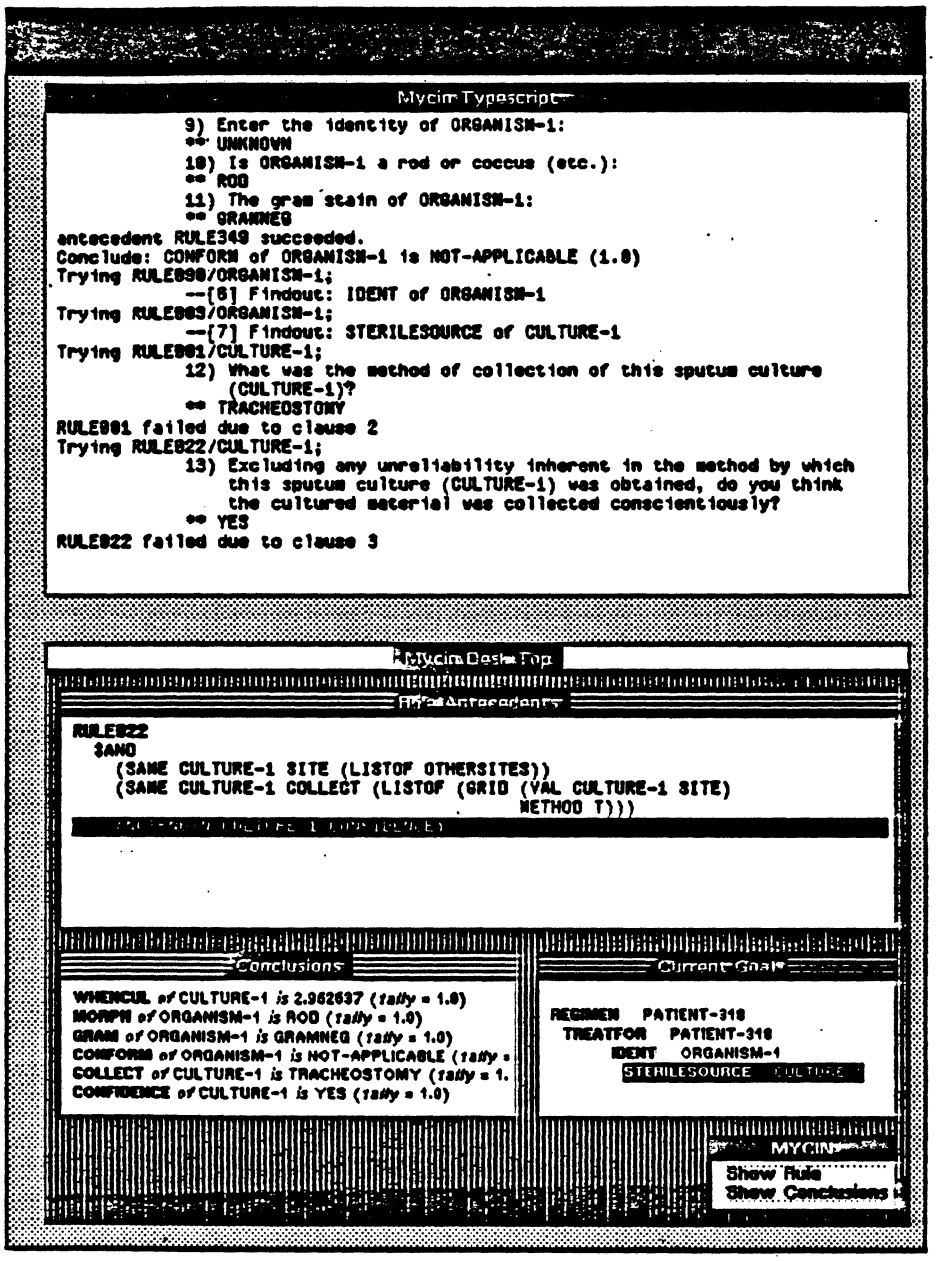

Mycin Typescript
9) Enter the identity of ORGANISM-1:
** UNKNOWN
10) Is ORGANISM-1 a rod or coccus (etc.):
** ROD
11) The gram stain of ORGANISM-1:
** GRAMNEG
antecedent RULE349 succeeded.
Conclude: CONFORM of ORGANISM-1 is NOT-APPLICABLE (1.0)
Trying RULE090/ORGANISM-1;
--[6] Findout: IDENT of ORGANISM-1
Trying RULE089/ORGANISM-1;
--[7] Findout: STERILESOURCE of CULTURE-1
Trying RULE001/CULTURE-1;
12) What was the method of collection of this sputum culture
(CULTURE-1)?
** TRACHEOSTOMY
RULE001 failed due to clause 2
Trying RULE022/CULTURE-1;
13) Excluding any unreliability inherent in the method by which
this sputum culture (CULTURE-1) was obtained, do you think
the cultured material was collected conscientiously?
** YES
RULE022 failed due to clause 3
Mycin Desk Top
Rule Antecedents
RULE022
$AND
(SAME CULTURE-1 SITE (LISTOF OTHERSITES))
(SAME CULTURE-1 COLLECT (LISTOF (GRID (VAL CULTURE-1 SITE)
METHOD T)))
Conclusions
WHENCUL of CULTURE-1 is 2.962637 (tally = 1.0)
MORPH of ORGANISM-1 is ROD (tally = 1.0)
GRAM of ORGANISM-1 is GRAMNEG (tally = 1.0)
CONFORM of ORGANISM-1 is NOT-APPLICABLE (tally =
COLLECT of CULTURE-1 is TRACHEOSTOMY (tally = 1.
CONFIDENCE of CULTURE-1 is YES (tally = 1.0)
Current Goal
REGIMEN PATIENT-318
TREATFOR PATIENT-318
IDENT ORGANISM-1
STERILESOURCE CULTURE
MYCIN
Show Rule
Show Conclusions

Figure 34. Concurrent Monitoring of a MYCIN Run, Scene 1

An Event is generated every time MYCIN attempts to apply a rule known to be relevant to the current situation. The concurrent monitor processes that Event and displays the antecedent of the rule in the Rule Antecedents Window. If the rule failed, the clause that evaluated to "false" is displayed with a reversed background. The tree of goals [the value of the variables CNTXT and PARM] is displayed in the Current Goal Window, with the current one highlighted. Note the use of fonts in these displays. More information can be included in these displays than in the teletype trace, for reasons discussed in the text -- compare the contents of the two traces.

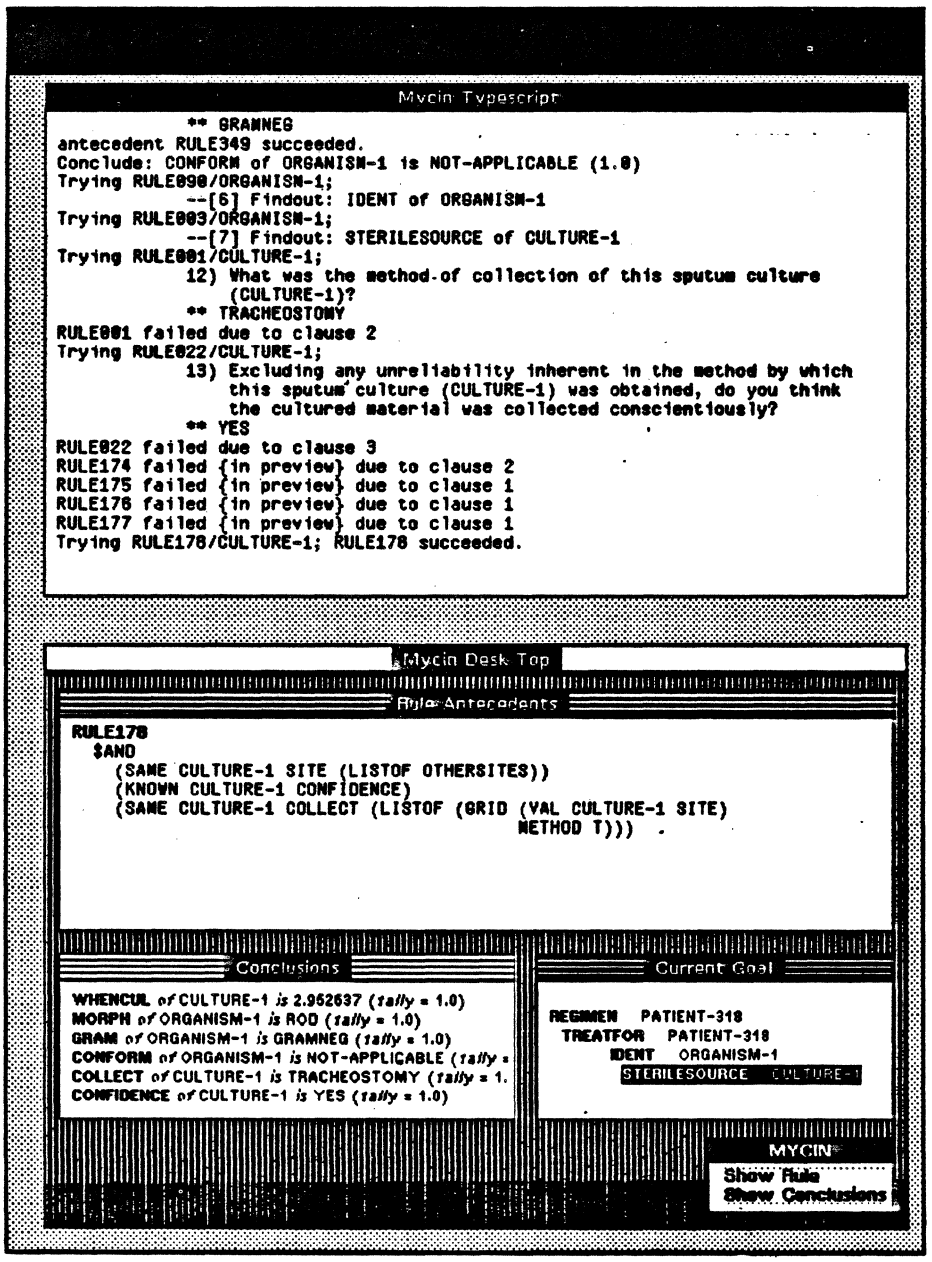

Mycin Typescript
** GRAMNEG
antecedent RULE349 succeeded.
Conclude: CONFORM of ORGANISM-1 is NOT-APPLICABLE (1.0)
Trying RULE090/ORGANISM-1;
--[6] Findout: IDENT of ORGANISM-1
Trying RULE003/ORGANISM-1;
--[7] Findout: STERILESOURCE of CULTURE-1
Trying RULE001/CULTURE-1;
12) What was the method of collection of this sputum culture
(CULTURE-1)?
** TRACHEOSTOMY
RULE001 failed due to clause 2
Trying RULE022/CULTURE-1;
13) Excluding any unreliability inherent in the method by which
this sputum culture (CULTURE-1) was obtained, do you think
the cultured material was collected conscientiously?
** YES
RULE022 failed due to clause 3
RULE174 failed {in preview} due to clause 2
RULE175 failed {in preview} due to clause 1
RULE176 failed {in preview} due to clause 1
RULE177 failed {in preview} due to clause 1
Trying RULE178/CULTURE-1; RULE178 succeeded.
Mycin Desk Top
Rule Antecedents
RULE178
$AND
(SAME CULTURE-1 SITE (LISTOF OTHERSITES))
(KNOWN CULTURE-1 CONFIDENCE)
(SAME CULTURE-1 COLLECT (LISTOF (GRID (VAL CULTURE-1 SITE)
METHOD T)))
Conclusions
WHENCUL of CULTURE-1 is 2.952637 (tally = 1.0)
MORPH of ORGANISM-1 is ROD (tally = 1.0)
GRAM of ORGANISM-1 is GRAMNEG (tally = 1.0)
CONFORM of ORGANISM-1 is NOT-APPLICABLE (tally =
COLLECT of CULTURE-1 is TRACHEOSTOMY (tally = 1.
CONFIDENCE of CULTURE-1 is YES (tally = 1.0)
Current Goal
REGIMEN PATIENT-318
TREATFOR PATIENT-318
IDENT ORGANISM-1
STERILESOURCE CULTURE-1
MYCIN
Show Rule
Show Conclusions

Figure 35. Concurrent Monitoring of a MYCIN Run, Scene 2

RULE178 has succeeded (no clause is highlighted). Successful rules generally produce conclusions, another point for Event generation. These conclusions are displayed in the Conclusions Window, which is scrolled to keep the most recent one at the bottom line. Note that the traces which MYCIN prints out interrupt the question and answer dialogue between the user and the system. Limiting this interference is one of the purposes of omitting details of the rules in the teletype trace. Sorting information into different windows makes including much of this detail feasible. The relative speeds of teletypes and display terminals are also a factor.

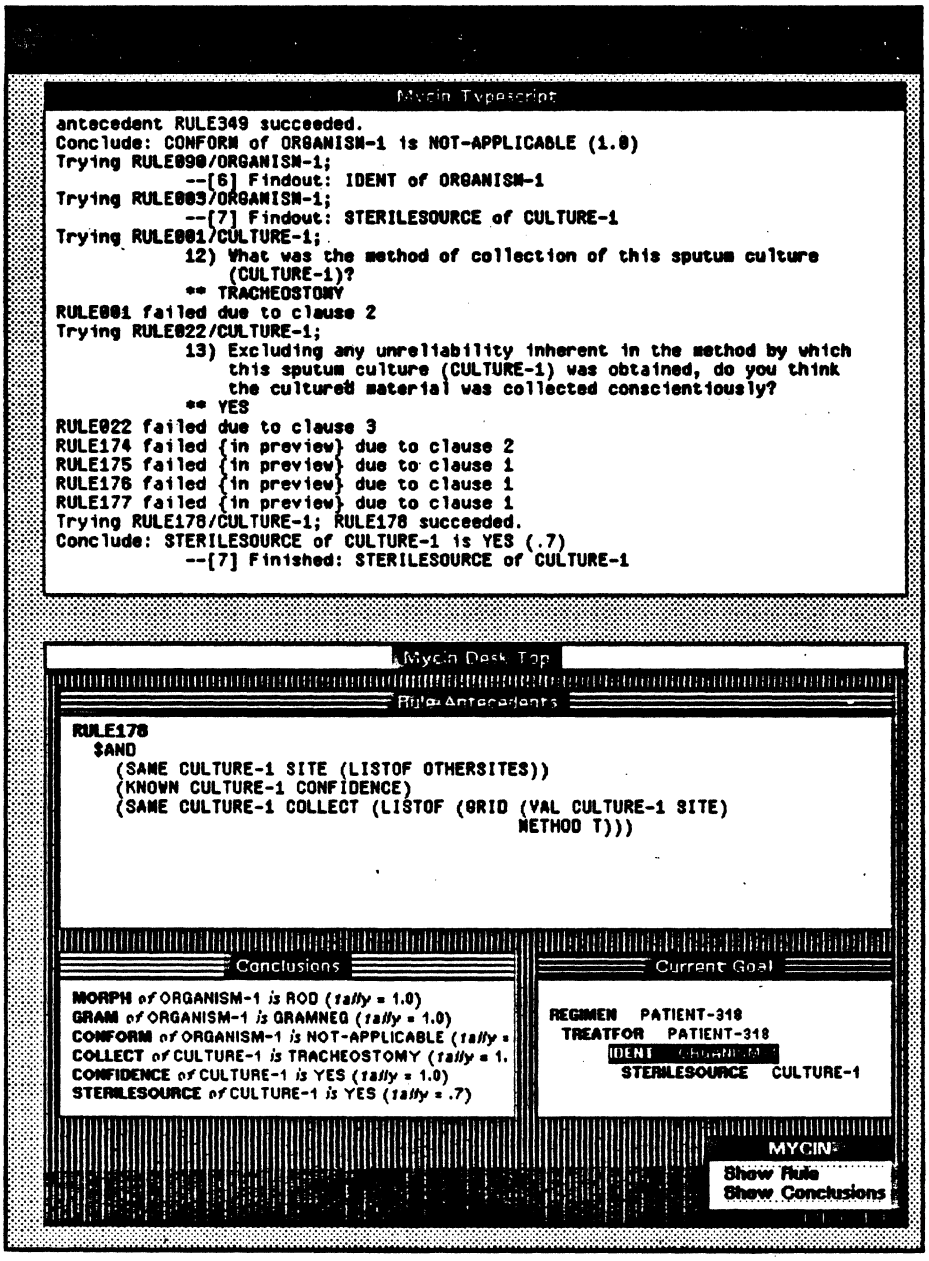

Mycin Typescript
antecedent RULE349 succeeded.
Conclude: CONFORM of ORGANISM-1 is NOT-APPLICABLE (1.0)
Trying RULE090/ORGANISM-1;
--[6] Findout: IDENT of ORGANISM-1
Trying RULE003/ORGANISM-1;
--[7] Findout: STERILESOURCE of CULTURE-1
Trying RULE001/CULTURE-1;
12) What was the method of collection of this sputum culture (CULTURE-1)?
** TRACHEOSTOMY
RULE001 failed due to clause 2
Trying RULE022/CULTURE-1;
13) Excluding any unreliability inherent in the method by which this sputum culture (CULTURE-1) was obtained, do you think the cultured material was collected conscientiously?
** YES
RULE022 failed due to clause 3
RULE174 failed (in preview) due to clause 2
RULE175 failed (in preview) due to clause 1
RULE176 failed (in preview) due to clause 1
RULE177 failed (in preview) due to clause 1
Trying RULE178/CULTURE-1; RULE178 succeeded.
Conclude: STERILESOURCE of CULTURE-1 is YES (.7)
--[7] Finished: STERILESOURCE of CULTURE-1
Mycin Desk Top
Rule Antecedents
RULE178
$AND
(SAME CULTURE-1 SITE (LISTOF OTHERSITES))
(KNOWN CULTURE-1 CONFIDENCE)
(SAME CULTURE-1 COLLECT (LISTOF (GRID (VAL CULTURE-1 SITE)
METHOD T)))
Conclusions
MORPH of ORGANISM-1 is ROD (tally = 1.0)
GRAM of ORGANISM-1 is GRAMNEG (tally = 1.0)
CONFORM of ORGANISM-1 is NOT-APPLICABLE (tally =
COLLECT of CULTURE-1 is TRACHEOSTOMY (tally = 1.
CONFIDENCE of CULTURE-1 is YES (tally = 1.0)
STERILESOURCE of CULTURE-1 is YES (tally = .7)
Current Goal
REGIMEN PATIENT-318
TREATFOR PATIENT-318
IDENT
STERILESOURCE CULTURE-1
MYCIN
Show Rule
Show Conclusions

Figure 36. Concurrent Monitoring of a MYCIN Run, Scene 3

As a result of the successful application of RULE178, a new conclusion has been made, pertaining to the parameter STERILESOURCE. This is shown in the Conclusions Window. The conclusion satisfied the STERILESOURCE of CULTURE-1 goal. Accordingly, an Event was generated and processed. The system continues to work on the parent goal of the one just satisfied, as indicated in the Current Goal Window. The MYCIN printout includes the depth of the goal in the implicit goal tree, printed as a number; this depth is indicated *analogically* in the display, through relative indentation of the text.

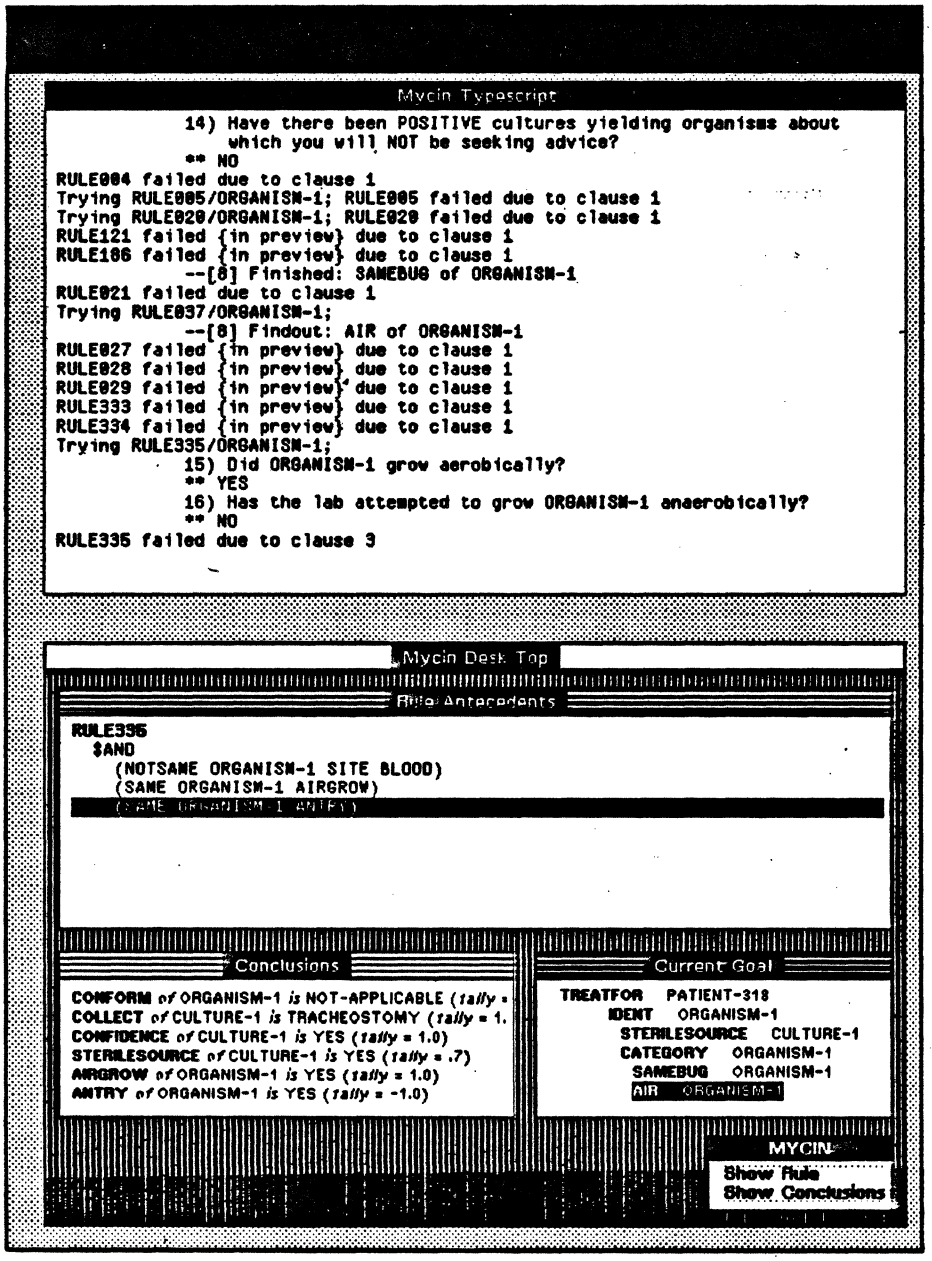

Mycin Typescript
14) Have there been POSITIVE cultures yielding organisms about which you will NOT be seeking advice?
** NO
RULE004 failed due to clause 1
Trying RULE005/ORGANISM-1; RULE005 failed due to clause 1
Trying RULE020/ORGANISM-1; RULE020 failed due to clause 1
RULE121 failed {in preview} due to clause 1
RULE186 failed {in preview} due to clause 1
--[0] Finished: SAMEBUG of ORGANISM-1
RULE021 failed due to clause 1
Trying RULE037/ORGANISM-1;
--[0] Findout: AIR of ORGANISM-1
RULE027 failed {in preview} due to clause 1
RULE028 failed {in preview} due to clause 1
RULE029 failed {in preview} due to clause 1
RULE333 failed {in preview} due to clause 1
RULE334 failed {in preview} due to clause 1
Trying RULE335/ORGANISM-1;
15) Did ORGANISM-1 grow aerobically?
** YES
16) Has the lab attempted to grow ORGANISM-1 anaerobically?
** NO
RULE335 failed due to clause 3
Mycin Desk Top
Rule Antecedents
RULE336
$AND
(NOTSAME ORGANISM-1 SITE BLOOD)
(SAME ORGANISM-1 AIRGROW)
Conclusions
CONFORM of ORGANISM-1 is NOT-APPLICABLE (tally =
COLLECT of CULTURE-1 is TRACHEOSTOMY (tally = 1.
CONFIDENCE of CULTURE-1 is YES (tally = 1.0)
STERILESOURCE of CULTURE-1 is YES (tally = .7)
AIRGROW of ORGANISM-1 is YES (tally = 1.0)
ANTRY of ORGANISM-1 is YES (tally = -1.0)
Current Goal
TREATFOR PATIENT-318
IDENT ORGANISM-1
STERILESOURCE CULTURE-1
CATEGORY ORGANISM-1
SAMEBUG ORGANISM-1
AIR ORGANISM-1
MYCIN
Show Rule
Show Conclusions

Figure 37. Concurrent Monitoring of a MYCIN Run, Scene 4

MYCIN continues to work. The amount of activity performed at this stage is growing, and the trace information is beginning to obscure the system dialogue in the Mycin Typescript Window. "Tally" in the Conclusions Window refers to "certainty factors", an essential part of the MYCIN reasoning scheme. Incidentally, the rules in the MYCIN system are phrased generally, using variable names such as PARM and CNTXT. In the display of the rule antecedents, the names of these variables have been replaced with their values.

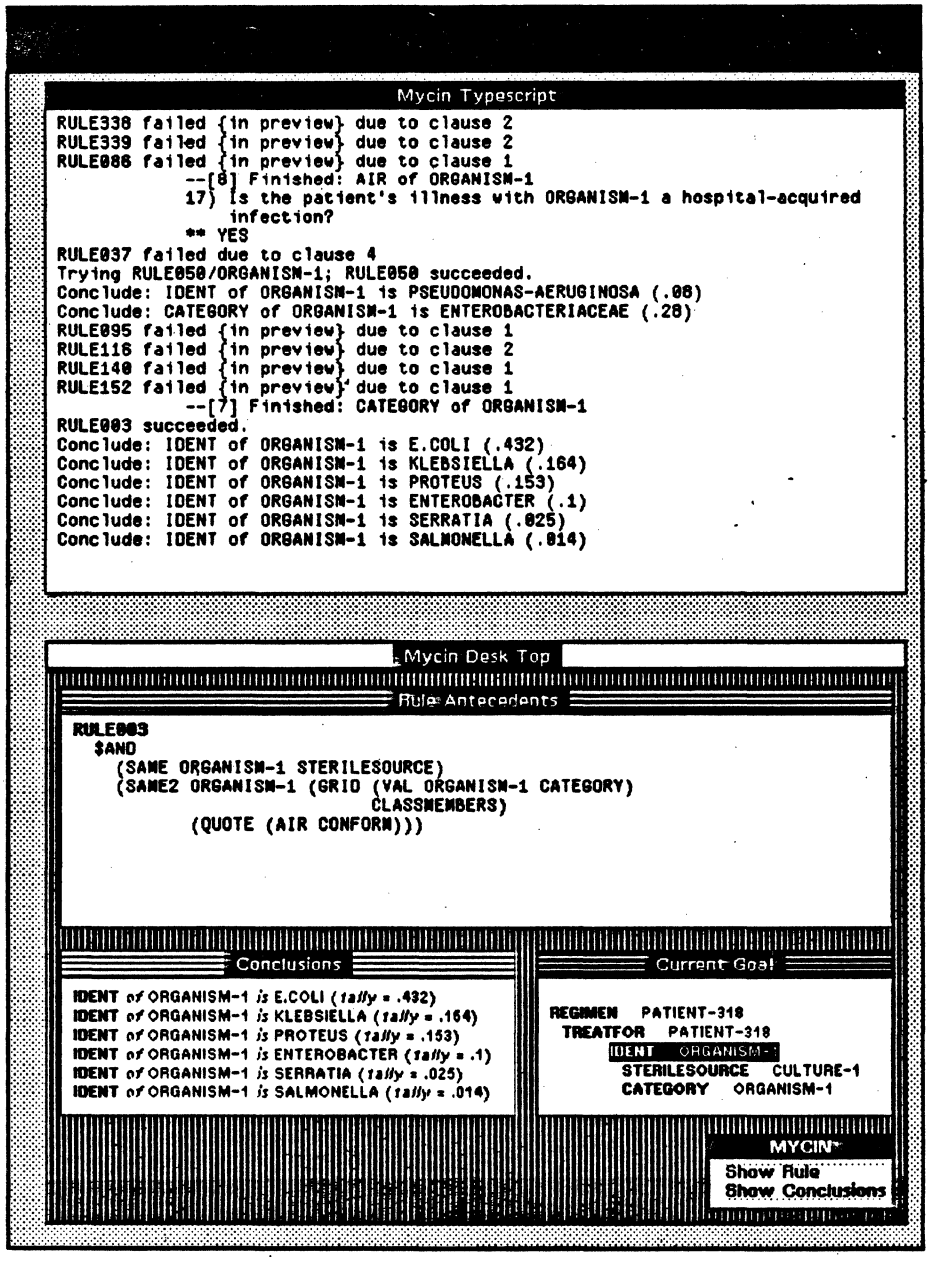

Mycin Typescript
RULE338 failed {in preview} due to clause 2
RULE339 failed {in preview} due to clause 2
RULE086 failed {in preview} due to clause 1
--[8] Finished: AIR of ORGANISM-1
17) Is the patient's illness with ORGANISM-1 a hospital-acquired infection?
** YES
RULE037 failed due to clause 4
Trying RULE050/ORGANISM-1; RULE050 succeeded.
Conclude: IDENT of ORGANISM-1 is PSEUDOMONAS-AERUGINOSA (.08)
Conclude: CATEGORY of ORGANISM-1 is ENTEROBACTERIACEAE (.28)
RULE095 failed {in preview} due to clause 1
RULE116 failed {in preview} due to clause 2
RULE140 failed {in preview} due to clause 1
RULE152 failed {in preview} due to clause 1
--[7] Finished: CATEGORY of ORGANISM-1
RULE003 succeeded.
Conclude: IDENT of ORGANISM-1 is E.COLI (.432)
Conclude: IDENT of ORGANISM-1 is KLEBSIELLA (.164)
Conclude: IDENT of ORGANISM-1 is PROTEUS (.153)
Conclude: IDENT of ORGANISM-1 is ENTEROBACTER (.1)
Conclude: IDENT of ORGANISM-1 is SERRATIA (.025)
Conclude: IDENT of ORGANISM-1 is SALMONELLA (.014)
Mycin Desk Top
Rule Antecedents
RULE003
$AND
(SAME ORGANISM-1 STERILESOURCE)
(SAME2 ORGANISM-1 (GRID (VAL ORGANISM-1 CATEGORY)
CLASSMEMBERS)
(QUOTE (AIR CONFORM)))
Conclusions
IDENT of ORGANISM-1 is E.COLI (tally = .432)
IDENT of ORGANISM-1 is KLEBSIELLA (tally = .164)
IDENT of ORGANISM-1 is PROTEUS (tally = .153)
IDENT of ORGANISM-1 is ENTEROBACTER (tally = .1)
IDENT of ORGANISM-1 is SERRATIA (tally = .025)
IDENT of ORGANISM-1 is SALMONELLA (tally = .014)
Current Goal
REGIMEN PATIENT-318
TREATFOR PATIENT-318
IDENT ORGANISM-1
STERILESOURCE CULTURE-1
CATEGORY ORGANISM-1
MYCIN
Show Rule
Show Conclusions

Figure 38. Concurrent Monitoring of a MYCIN Run, Scene 5

MYCIN has determined several possible identities for ORGANISM-1, with varying degrees of certainty. The display in the Conclusions Window is similar to the MYCIN teletype trace, but the font information helps make the former more readable than the latter. In the middle of the Mycin Typescript Window, somewhat obscured by the surrounding lines, there are two other conclusions. These conclusions are not showing in the display either, but there, a quick scrolling operation will bring them into view, contiguous with the conclusions that immediately follow it. Sorting different trace streams into their own windows both disentangles and unifies them.

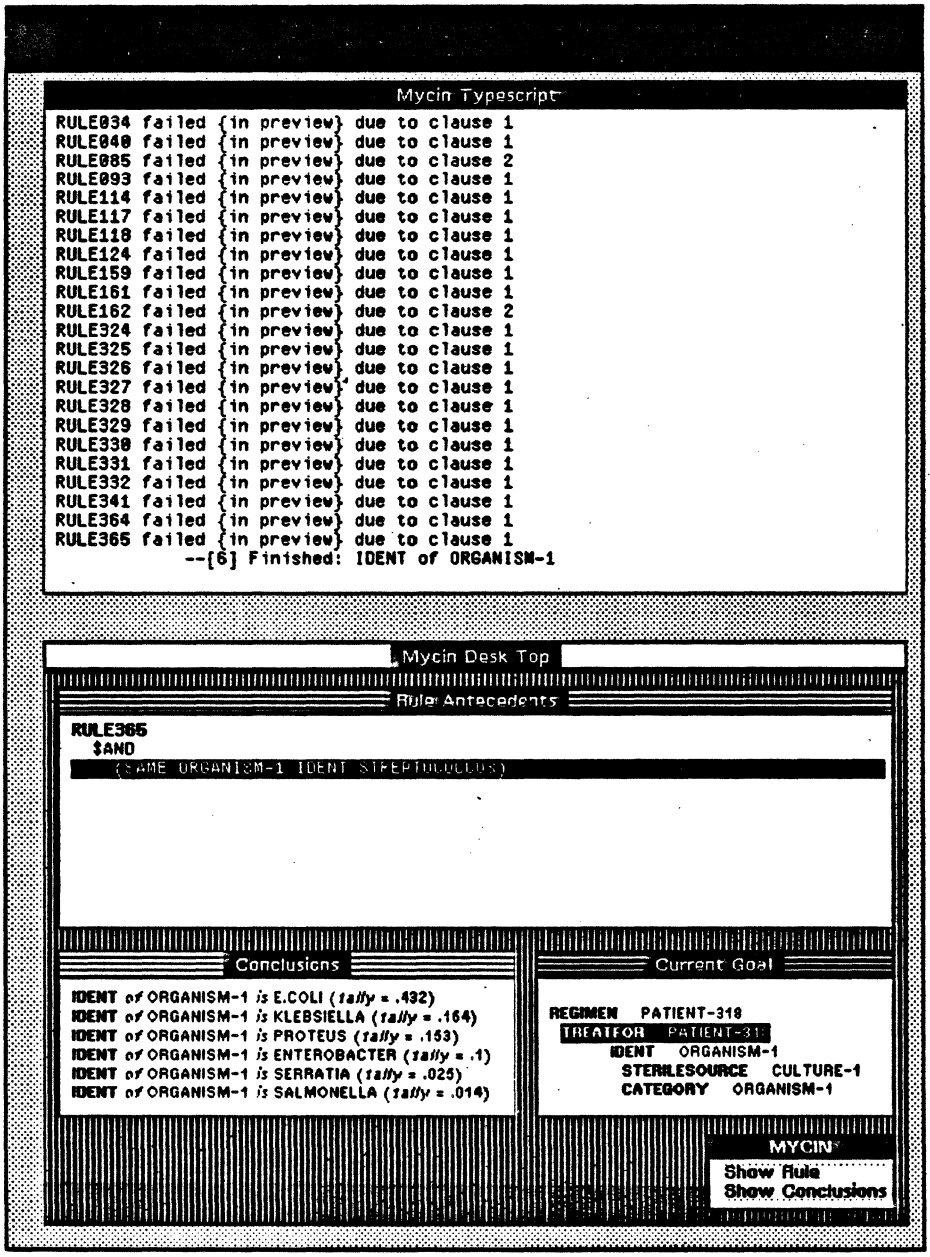

Mycin Typescript
RULE034 failed {in preview} due to clause 1
RULE040 failed {in preview} due to clause 1
RULE085 failed {in preview} due to clause 2
RULE093 failed {in preview} due to clause 1
RULE114 failed {in preview} due to clause 1
RULE117 failed {in preview} due to clause 1
RULE118 failed {in preview} due to clause 1
RULE124 failed {in preview} due to clause 1
RULE159 failed {in preview} due to clause 1
RULE161 failed {in preview} due to clause 1
RULE162 failed {in preview} due to clause 2
RULE324 failed {in preview} due to clause 1
RULE325 failed {in preview} due to clause 1
RULE326 failed {in preview} due to clause 1
RULE327 failed {in preview} due to clause 1
RULE328 failed {in preview} due to clause 1
RULE329 failed {in preview} due to clause 1
RULE330 failed {in preview} due to clause 1
RULE331 failed {in preview} due to clause 1
RULE332 failed {in preview} due to clause 1
RULE341 failed {in preview} due to clause 1
RULE364 failed {in preview} due to clause 1
RULE365 failed {in preview} due to clause 1
--[6] Finished: IDENT of ORGANISM-1
Mycin Desk Top
Rule Antecedents
RULE365
$AND
(SAME ORGANISM-1 IDENT STREPTOCOCCUS)
Conclusions
IDENT of ORGANISM-1 is E.COLI (tally = .432)
IDENT of ORGANISM-1 is KLEBSIELLA (tally = .164)
IDENT of ORGANISM-1 is PROTEUS (tally = .153)
IDENT of ORGANISM-1 is ENTEROBACTER (tally = .1)
IDENT of ORGANISM-1 is SERRATIA (tally = .025)
IDENT of ORGANISM-1 is SALMONELLA (tally = .014)
Current Goal
REGIMEN PATIENT-318
TREATFOR PATIENT-318
IDENT ORGANISM-1
STERILESOURCE CULTURE-1
CATEGORY ORGANISM-1
MYCIN
Show Rule
Show Conclusions

Figure 39. Concurrent Monitoring of a MYCIN Run, Scene 6

In this scene, the Mycin Typescript Window contains much repetitious information which the user is unlikely to find helpful. Only the most recent rule appears in the Rule Antecendent Window, so the screen is not cluttered with this information. If the user should want it, however, he can always retrieve it by scrolling. Another way the user can examine old information is by using the operations on the EDIT MENU that search selected text for a typed-in key string. Thus, the user can find all the attempted applications of a designated rule, or all those which used a particular parameter, etc., and similarly for the other windows.

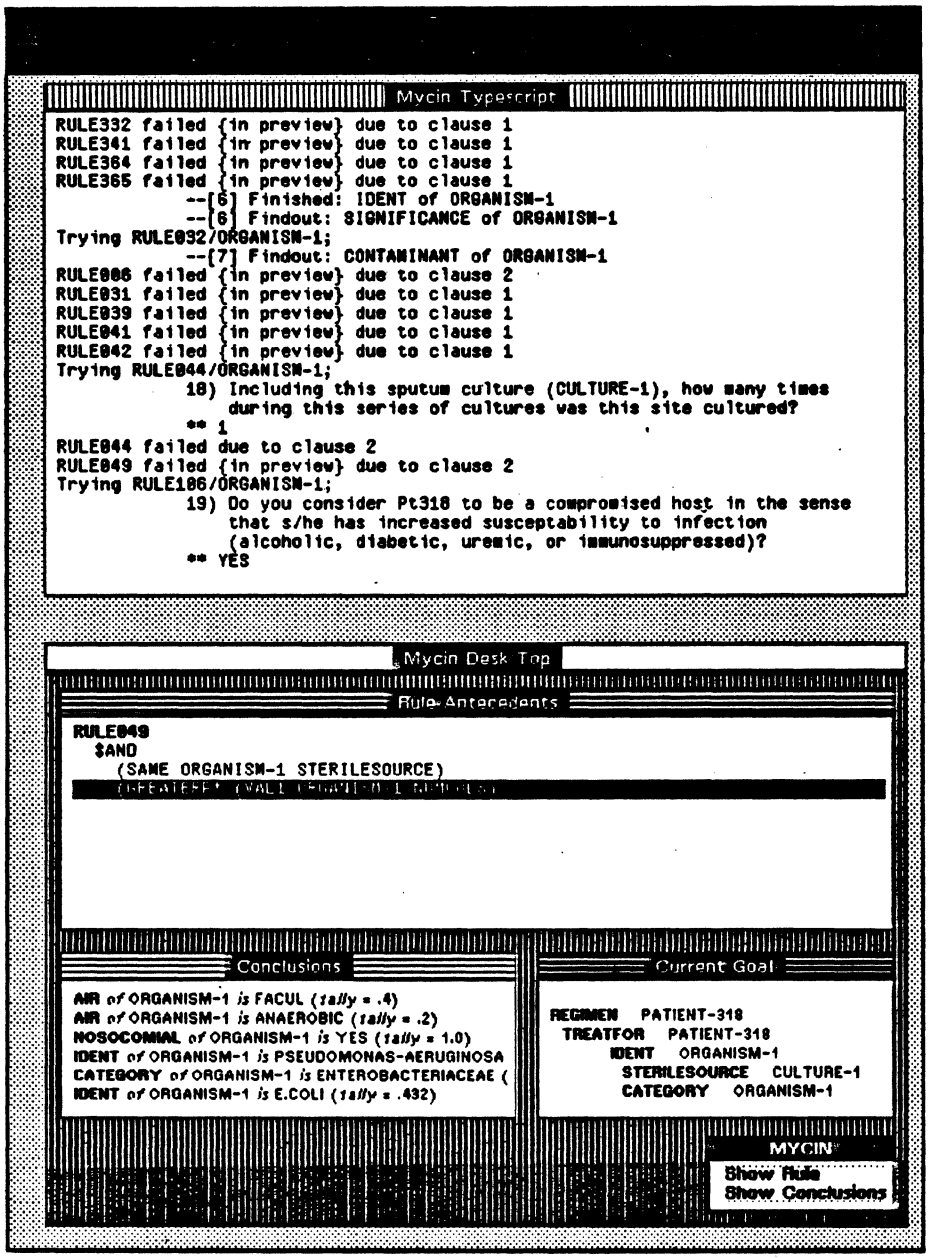

Mycin Typescript
RULE332 failed {in preview} due to clause 1
RULE341 failed {in preview} due to clause 1
RULE364 failed {in preview} due to clause 1
RULE365 failed {in preview} due to clause 1
--[6] Finished: IDENT of ORGANISM-1
--[6] Findout: SIGNIFICANCE of ORGANISM-1
Trying RULE092/ORGANISM-1;
--[7] Findout: CONTAMINANT of ORGANISM-1
RULE006 failed {in preview} due to clause 2
RULE031 failed {in preview} due to clause 1
RULE039 failed {in preview} due to clause 1
RULE041 failed {in preview} due to clause 1
RULE042 failed {in preview} due to clause 1
Trying RULE044/ORGANISM-1;
18) Including this sputum culture (CULTURE-1), how many times
during this series of cultures was this site cultured?
** 1
RULE044 failed due to clause 2
RULE049 failed {in preview} due to clause 2
Trying RULE186/ORGANISM-1;
19) Do you consider Pt318 to be a compromised host in the sense
that s/he has increased susceptability to infection
(alcoholic, diabetic, uremic, or immunosuppressed)?
** YES
Mycin Desk Top
Rule Antecedents
RULE049
$AND
(SAME ORGANISM-1 STERILESOURCE)
Conclusions
AIR of ORGANISM-1 is FACUL (tally = .4)
AIR of ORGANISM-1 is ANAEROBIC (tally = .2)
NOSOCOMIAL of ORGANISM-1 is YES (tally = 1.0)
IDENT of ORGANISM-1 is PSEUDOMONAS-AERUGINOSA
CATEGORY of ORGANISM-1 is ENTEROBACTERIACEAE (
IDENT of ORGANISM-1 is E.COLI (tally = .432)
Current Goal
REGIMEN PATIENT-318
TREATFOR PATIENT-318
IDENT ORGANISM-1
STERILESOURCE CULTURE-1
CATEGORY ORGANISM-1
MYCIN
Show Rule
Show Conclusions

Figure 40. Concurrent Monitoring of a MYCIN Run, Scene 7

The execution has been suspended by buttoning the caption area of the Mycin Typescript Area (which is why it is now the CURRENT WINDOW). Execution can be resumed by simply buttoning that caption again. The Conclusions Window has been scrolled back to an earlier point. Between this scene and the previous one, the Current Goal extended below CATEGORY ORGANISM-1. This can be determined by noticing that no Goal is currently highlighted. The window had been scrolled back to its beginning just before the Conclusions Window was scrolled.

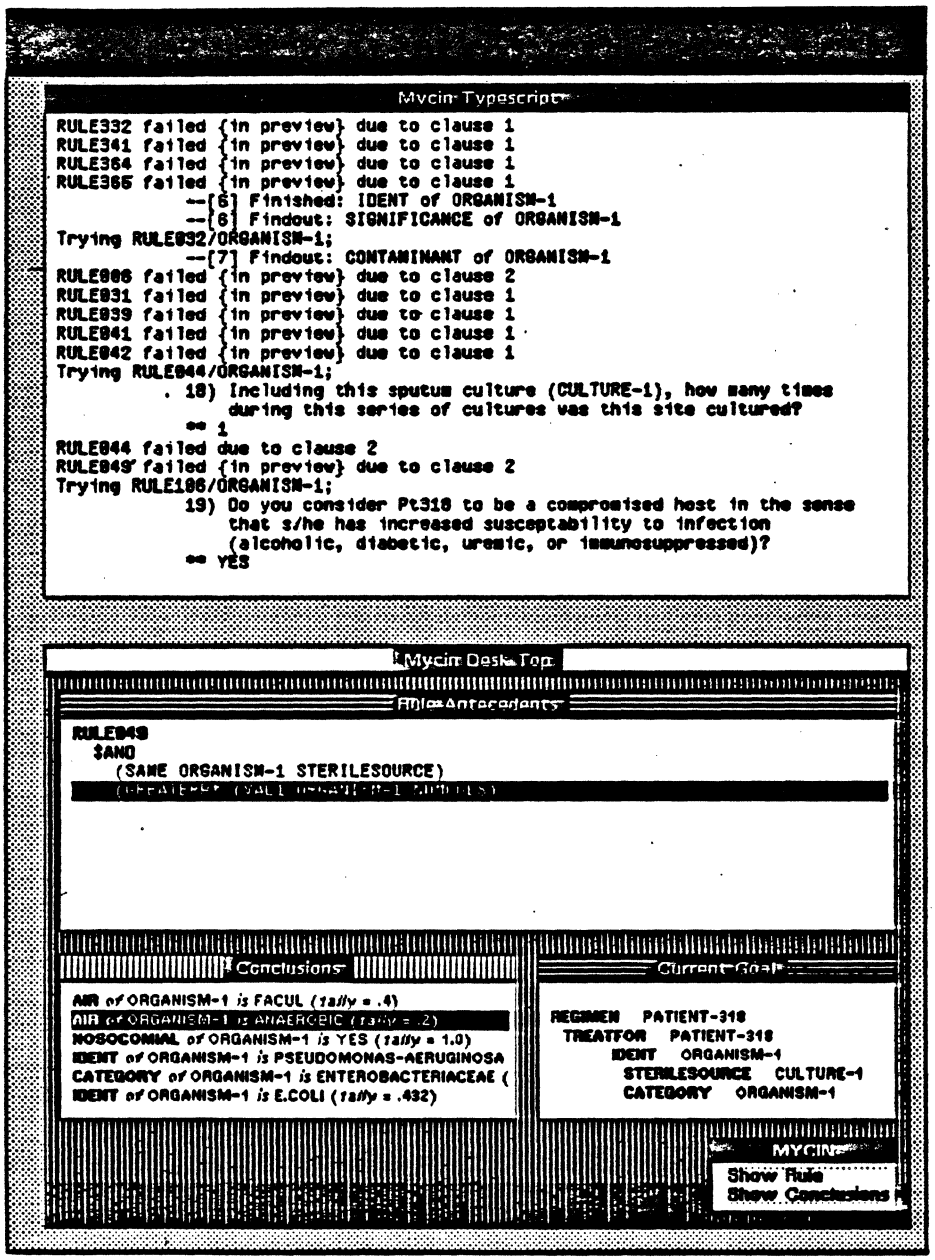

Mycin Typescript
RULE332 failed {in preview} due to clause 1
RULE341 failed {in preview} due to clause 1
RULE364 failed {in preview} due to clause 1
RULE365 failed {in preview} due to clause 1
--[6] Finished: IDENT of ORGANISM-1
--[6] Findout: SIGNIFICANCE of ORGANISM-1
Trying RULE092/ORGANISM-1;
--[7] Findout: CONTAMINANT of ORGANISM-1
RULE086 failed {in preview} due to clause 2
RULE031 failed {in preview} due to clause 1
RULE039 failed {in preview} due to clause 1
RULE041 failed {in preview} due to clause 1
RULE042 failed {in preview} due to clause 1
Trying RULE044/ORGANISM-1;
. 18) Including this sputum culture (CULTURE-1), how many times during this series of cultures was this site cultured?
** 1
RULE044 failed due to clause 2
RULE049 failed {in preview} due to clause 2
Trying RULE186/ORGANISM-1;
19) Do you consider Pt318 to be a compromised host in the sense that s/he has increased susceptability to infection (alcoholic, diabetic, uremic, or immunosuppressed)?
** YES
Mycin Desk Top
RULE049
$AND
(SAME ORGANISM-1 STERILESOURCE)
Conclusions
AIR of ORGANISM-1 is FACUL (tally = .4)
AIR of ORGANISM-1 is ANAEROBIC (tally = .2)
NOSOCOMIAL of ORGANISM-1 is YES (tally = 1.0)
IDENT of ORGANISM-1 is PSEUDOMONAS-AERUGINOSA
CATEGORY of ORGANISM-1 is ENTEROBACTERIACEAE (
IDENT of ORGANISM-1 is E.COLI (tally = .432)
Current Goal
REGIMEN PATIENT-318
TREATFOR PATIENT-318
IDENT ORGANISM-1
STERILESOURCE CULTURE-1
CATEGORY ORGANISM-1
MYCIN
Show Rule
Show Conclusions

Figure 41. Concurrent Monitoring of a MYCIN Run, Scene 8

Using the ordinary DLISP selection mechanism, a conclusion has been selected (making the Conclusions Window the CURRENT WINDOW).

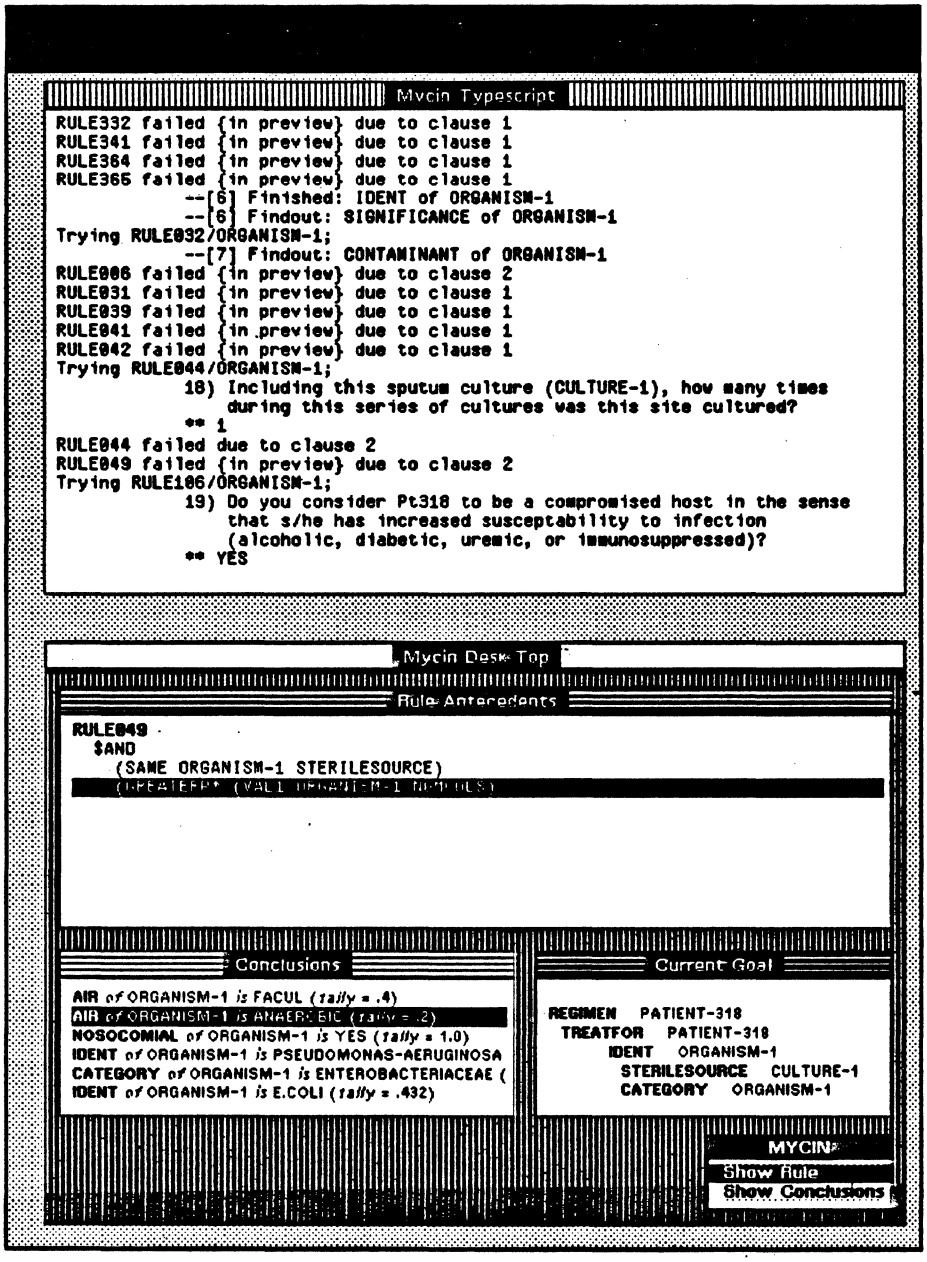

Mycin Typescript
RULE332 failed {in preview} due to clause 1
RULE341 failed {in preview} due to clause 1
RULE364 failed {in preview} due to clause 1
RULE365 failed {in preview} due to clause 1
--[6] Finished: IDENT of ORGANISM-1
--[6] Findout: SIGNIFICANCE of ORGANISM-1
Trying RULE032/ORGANISM-1;
--[7] Findout: CONTAMINANT of ORGANISM-1
RULE006 failed {in preview} due to clause 2
RULE031 failed {in preview} due to clause 1
RULE039 failed {in preview} due to clause 1
RULE041 failed {in preview} due to clause 1
RULE042 failed {in preview} due to clause 1
Trying RULE044/ORGANISM-1;
18) Including this sputum culture (CULTURE-1), how many times
during this series of cultures was this site cultured?
** 1
RULE044 failed due to clause 2
RULE049 failed {in preview} due to clause 2
Trying RULE106/ORGANISM-1;
19) Do you consider Pt318 to be a compromised host in the sense
that s/he has increased susceptability to infection
(alcoholic, diabetic, uremic, or immunosuppressed)?
** YES
Mycin Desk Top
Rule Antecedents
RULE049
$AND
(SAME ORGANISM-1 STERILESOURCE)
Conclusions
AIR of ORGANISM-1 is FACUL (tally = .4)
NOSOCOMIAL of ORGANISM-1 is YES (tally = 1.0)
IDENT of ORGANISM-1 is PSEUDOMONAS-AERUGINOSA
CATEGORY of ORGANISM-1 is ENTEROBACTERIACEAE (
IDENT of ORGANISM-1 is E.COLI (tally = .432)
Current Goal
REGIMEN PATIENT-318
TREATFOR PATIENT-318
IDENT ORGANISM-1
STERILESOURCE CULTURE-1
CATEGORY ORGANISM-1
MYCIN
Show Rule
Show Conclusions

Figure 42. Concurrent Monitoring of a MYCIN Run, Scene 9

The Show Rule operation on the MYCIN MENU has been buttoned. The construction of new menus is a straightforward matter using standard DLISP functions and data structures.

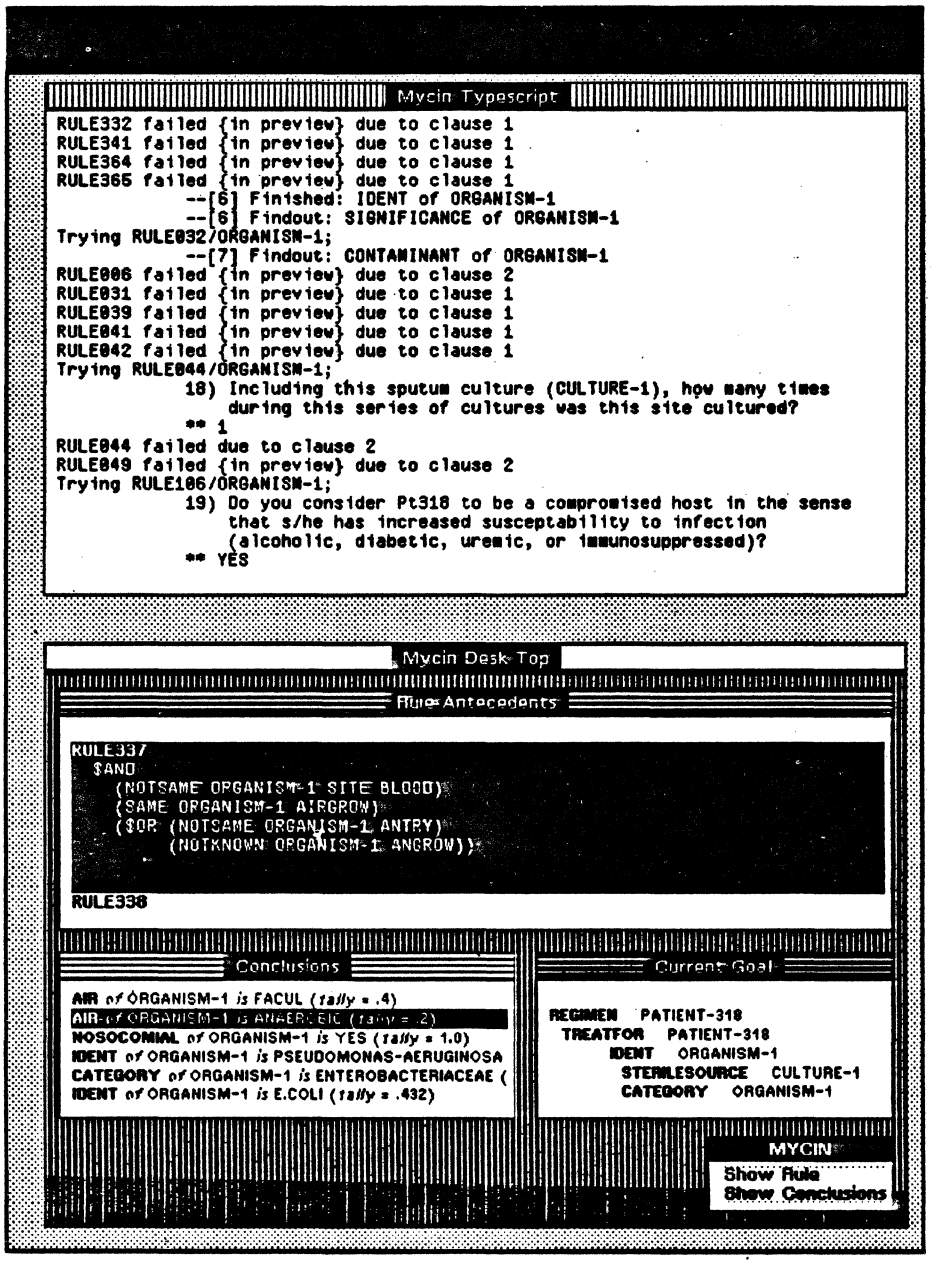

Mycin Typescript
RULE332 failed {in preview} due to clause 1
RULE341 failed {in preview} due to clause 1
RULE364 failed {in preview} due to clause 1
RULE365 failed {in preview} due to clause 1
--[6] Finished: IDENT of ORGANISM-1
--[6] Findout: SIGNIFICANCE of ORGANISM-1
Trying RULE032/ORGANISM-1;
--[7] Findout: CONTAMINANT of ORGANISM-1
RULE006 failed {in preview} due to clause 2
RULE031 failed {in preview} due to clause 1
RULE039 failed {in preview} due to clause 1
RULE041 failed {in preview} due to clause 1
RULE042 failed {in preview} due to clause 1
Trying RULE044/ORGANISM-1;
18) Including this sputum culture (CULTURE-1), how many times during this series of cultures was this site cultured?
** 1
RULE044 failed due to clause 2
RULE049 failed {in preview} due to clause 2
Trying RULE106/ORGANISM-1;
19) Do you consider Pt318 to be a compromised host in the sense that s/he has increased susceptability to infection (alcoholic, diabetic, uremic, or immunosuppressed)?
** YES
Mycin Desk Top
Rule Antecedents
RULE337
$AND
(NOTSAME ORGANISM-1 SITE BLOOD)
(SAME ORGANISM-1 AIRGROW)
($OR (NOTSAME ORGANISM-1 ANTRY)
(NOTKNOWN ORGANISM-1 ANGROW))
RULE338
Conclusions
AIR of ORGANISM-1 is FACUL (tally = .4)
AIR of ORGANISM-1 is ANAEROBIC (tally = .2)
NOSOCOMIAL of ORGANISM-1 is YES (tally = 1.0)
IDENT of ORGANISM-1 is PSEUDOMONAS-AERUGINOSA
CATEGORY of ORGANISM-1 is ENTEROBACTERIACEAE (
IDENT of ORGANISM-1 is E.COLI (tally = .432)
Current Goal
REGIMEN PATIENT-318
TREATFOR PATIENT-318
IDENT ORGANISM-1
STERILESOURCE CULTURE-1
CATEGORY ORGANISM-1
MYCIN
Show Rule
Show Conclusions

Figure 43. Concurrent Monitoring of a MYCIN Run, Scene 10

The Show Rule operation has been performed, causing the Rule Antecedents Window to be scrolled back to the rule which caused the selected conclusion. The operation also selects this rule. Note that the beginning of a new rule shows at the bottom line of the Rule Antecedents Window. When the user scrolls the window, it is likely that more than one rule will appear. Only one rule showed in each of the previous scenes, because the dynamic monitor automatically scrolls the Rule Antecedents Window to eliminate all but the current rule. This contributes to the clarity of the display, while still retaining the old information.

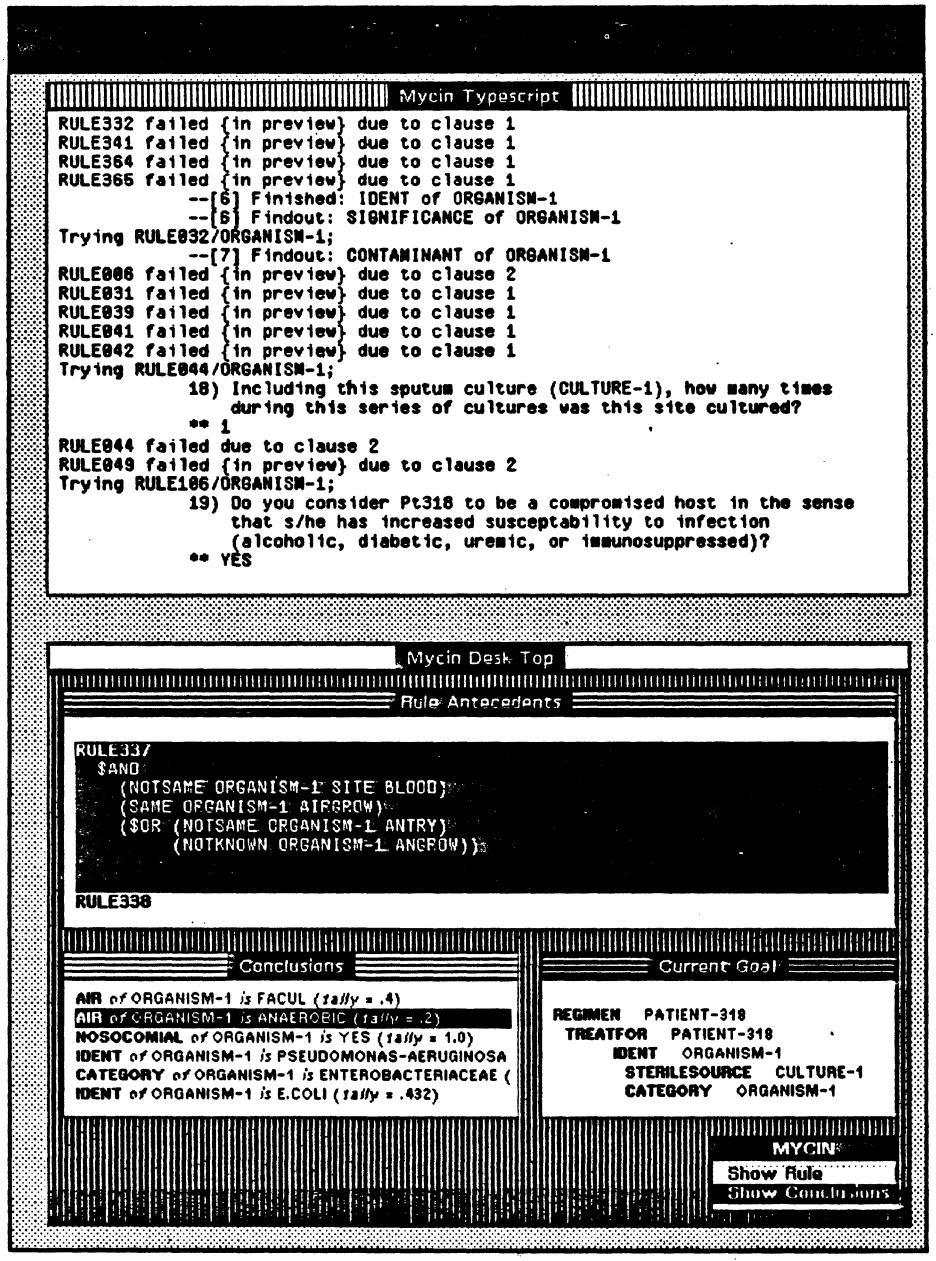

Mycin Typescript
RULE332 failed {in preview} due to clause 1
RULE341 failed {in preview} due to clause 1
RULE364 failed {in preview} due to clause 1
RULE365 failed {in preview} due to clause 1
--[6] Finished: IDENT of ORGANISM-1
--[6] Findout: SIGNIFICANCE of ORGANISM-1
Trying RULE032/ORGANISM-1;
--[7] Findout: CONTAMINANT of ORGANISM-1
RULE006 failed {in preview} due to clause 2
RULE031 failed {in preview} due to clause 1
RULE039 failed {in preview} due to clause 1
RULE041 failed {in preview} due to clause 1
RULE042 failed {in preview} due to clause 1
Trying RULE044/ORGANISM-1;
18) Including this sputum culture (CULTURE-1), how many times during this series of cultures was this site cultured?
** 1
RULE044 failed due to clause 2
RULE049 failed {in preview} due to clause 2
Trying RULE106/ORGANISM-1;
19) Do you consider Pt318 to be a compromised host in the sense that s/he has increased susceptability to infection (alcoholic, diabetic, uremic, or immunosuppressed)?
** YES
Mycin Desk Top
Rule Antecedents
RULE337
$AND
(NOTSAME ORGANISM-1 SITE BLOOD)
(SAME ORGANISM-1 AIRGROW)
($OR (NOTSAME ORGANISM-1 ANTRY)
(NOTKNOWN ORGANISM-1 ANGROW))
RULE338
Conclusions
AIR of ORGANISM-1 is FACUL (tally = .4)
AIR of ORGANISM-1 is ANAEROBIC (tally = .2)
NOSOCOMIAL of ORGANISM-1 is YES (tally = 1.0)
IDENT of ORGANISM-1 is PSEUDOMONAS-AERUGINOSA
CATEGORY of ORGANISM-1 is ENTEROBACTERIACEAE (
IDENT of ORGANISM-1 is E.COLI (tally = .432)
Current Goal
REGIMEN PATIENT-318
TREATFOR PATIENT-318
IDENT ORGANISM-1
STERILESOURCE CULTURE-1
CATEGORY ORGANISM-1
MYCIN
Show Rule
Show Conclusions

Figure 44. Concurrent Monitoring of a MYCIN Run, Scene 11

With a rule selected (either by the user or by a Show Rule operation), the Show Conclusions operation can be performed. It has been buttoned in this scene, using the rule selected by the previous Show Rule operation. Note that only successful rules can produce conclusions (though not all do), so that the use of inverted displays for this kind of selection does not conflict with highlighting the failed clause of an unsuccessful rule.

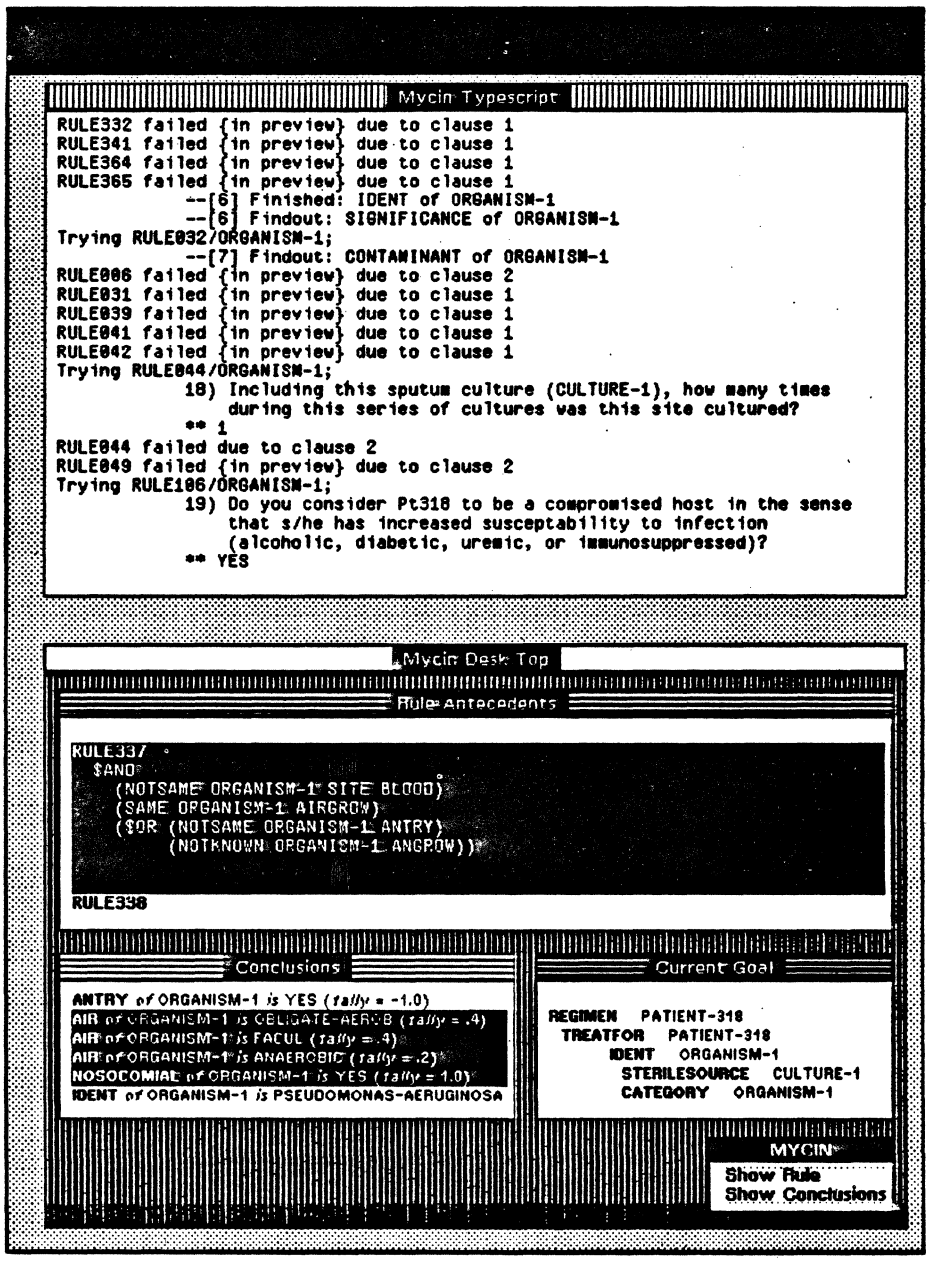

Mycin Typescript
RULE332 failed {in preview} due to clause 1
RULE341 failed {in preview} due to clause 1
RULE364 failed {in preview} due to clause 1
RULE365 failed {in preview} due to clause 1
--[6] Finished: IDENT of ORGANISM-1
--[6] Findout: SIGNIFICANCE of ORGANISM-1
Trying RULE032/ORGANISM-1;
--[7] Findout: CONTAMINANT of ORGANISM-1
RULE006 failed {in preview} due to clause 2
RULE031 failed {in preview} due to clause 1
RULE039 failed {in preview} due to clause 1
RULE041 failed {in preview} due to clause 1
RULE042 failed {in preview} due to clause 1
Trying RULE044/ORGANISM-1;
18) Including this sputum culture (CULTURE-1), how many times during this series of cultures was this site cultured?
** 1
RULE044 failed due to clause 2
RULE049 failed {in preview} due to clause 2
Trying RULE106/ORGANISM-1;
19) Do you consider Pt318 to be a compromised host in the sense that s/he has increased susceptability to infection (alcoholic, diabetic, uremic, or immunosuppressed)?
** YES
Mycin Desk Top
Rule Antecedents
RULE337
$AND
(NOTSAME ORGANISM-1 SITE BLOOD)
(SAME ORGANISM-1 AIRGROW)
($OR (NOTSAME ORGANISM-1 ANTRY)
(NOTKNOWN ORGANISM-1 ANGROW))
RULE338
Conclusions
ANTRY of ORGANISM-1 is YES (tally = -1.0)
AIR of ORGANISM-1 is OBLIGATE-AEROB (tally = .4)
AIR of ORGANISM-1 is FACUL (tally = .4)
AIR of ORGANISM-1 is ANAEROBIC (tally = .2)
NOSOCOMIAL of ORGANISM-1 is YES (tally = 1.0)
IDENT of ORGANISM-1 is PSEUDOMONAS-AERUGINOSA
Current Goal
REGIMEN PATIENT-318
TREATFOR PATIENT-318
IDENT ORGANISM-1
STERILESOURCE CULTURE-1
CATEGORY ORGANISM-1
MYCIN
Show Rule
Show Conclusions

Figure 45. Concurrent Monitoring of a MYCIN Run, Scene 12

The Conclusions Window has been appropriately scrolled by the Show Conclusions operation. The conclusions resulting from the selected rule are selected. Note that in this case there were several. Here is another example of answering questions *about* system activity rather than simply showing it. The user of an Event-oriented system should be able to obtain, in a direct manner, information about related Events. A facility such as the one illustrated here would be frequently utilized, either in the midst of an execution (as in this illustration), or as part of a post-execution investigation of the Events Space.

Chapter 10

Conclusion

This work began with the observation that traditional tools and techniques for the debugging of computer programs are inadequate in the context of complex systems such as those typical of Artificial Intelligence research. The challenge addressed was the development of a new approach to the design of monitoring facilities appropriate for AI systems. As a foundation for this enterprise, the nature of debugging activity, the history and features of debugging technology, the characteristics of complex systems, and some relevant features of human psychology and physiology were all explored. The proposed solution unites a new approach to the design of monitoring facilities, a generalized method of producing descriptions of system activity, and a set of graphically-oriented tools and techniques. In addition to summarizing these results, this chapter examines an important theme illuminated but insufficiently pursued in this work which would be an excellent direction for future research.

A. Summary

The first component of this work's proposals is an approach that involves three main thrusts: high-level orientation, graphical presentation of analogic information, and recognition of differences between discrete states and active processes. The first of these begins with the observation that systems (such as AI representation systems) are at a hierarchical level above ordinary programs and programming languages. At the minimum, traditional tools and techniques must be upgraded and reoriented to address system-level structures and operations. Further, the quantity and nature of the information required by the user of an AI system is such that simply showing the user full, unprocessed details about his system's operation would be infeasible and inappropriate. A concept called meta-monitoring is proposed: the user should be shown information *about* system activity rather than data directly *reflecting* the low-level details of its operation. In addition, information should be presented in response to user requests or in anticipation of user needs, instead of mechanically relaying all facets of all computational activity. Finally, data about low-level phenomena must be compiled into structures at a higher conceptual level to ease the user's chore of interpreting monitoring information.

The second aspect of the approach to monitoring is based on the advantages of visual modes of information presentation for many purposes. The graphical presentation of analogic information contributes to effective high-level communication and facilitates a greater flow of intelligible data. The third part of the approach is the recognition of important differences between the nature of discrete computational states and that of dynamic computational processes. These differences must be taken into account in the design of system-level monitoring facilities so that the user is assisted in his attempts to gain an understanding of the system's dynamic behavior and not just of its static features.

A generalized method for generating descriptions of system activity complements and grounds this approach to monitoring. It is based on there being in each system a small set of fundamental structures and operations on them, each of which has a small number of key components. Each time one of these fundamental operations occurs, an Event is generated which describes the activity in terms of fundamental structures and the key components of the structures and operations. In this way, a fairly complete, yet nonetheless high-level description of system activity is efficiently produced. The general method is easily instantiated for any particular system: that system's fundamental structures and operations are determined, along with their key components, and the system code is altered to generate Events at appropriate times. The stream of Events then becomes the raw material for a monitor. This information can be used in a variety of ways: for concurrent monitoring of important features of system activity; for "movie" replays of system activity at speeds, on levels of detail, and from viewpoints controlled by the user; for post-execution investigations of the recorded Event history; and, as a bonus, for internal reasoning within the system itself.

Specific display-based tools and techniques support the embodiment of the proposed approach and generalized descriptive method in system-oriented monitors. The utility of a high-level facility for graphics programming integrated with a standard programming language was demonstrated through its employment in the implementation of various monitoring tools. Graphical tree displays, bargraphs (both static and dynamic), multi-window organization of the screen, and font and background information used to distinguish various aspects and states of display components are examples of display-based tools which contribute to fulfilling the theoretical design requirements proposed here. The value of the theoretical approach and the tools and techniques developed in accordance with its dictates were demonstrated in the experimental monitoring facilities constructed in the course of this work. The advantages of separating a system and its associated monitor so that they are essentially independent of each other was also explored in the experimentation.

B. An Unfulfilled Theme

There is a set of loose ends left untied in this work, mostly in the form of suggestive ideas presented in examples but inadequately pursued in the research. In addition, there were other stray threads which had been extirpated earlier, as it became clear that satisfactory progress had not been made towards integrating them into the fabric of this work. A common theme unifies these and defines a coherent direction to be pursued in some future research. The theme is that of system self-understanding, and its inadequate development in this work prevented the full development of a central principle of the proposed approach to monitoring: system compilation of lower-level information into higher-level constructs through the detection of patterns in the lower level phenomena. This section examines this theme and shows the important direction in which it leads.

The theme of system self-understanding surfaced at several points in earlier chapters. One program-level example involved arrays. In the final part of Chapter 6, Section A it was suggested that rather than printing out all the elements of an array, it would often be more appropriate to indicate that the array was still initialized or that all its values were -1 (perhaps with a few exceptions). Programming languages should provide built-in mechanisms for the formatted printing of all structured objects in the language. The user should not have to continually be specifying printing iterations and formats. In like manner, programming languages ought to have operations such as INITIALIZE ARRAY. The cumulative savings of the time and effort which programmers waste writing iteration loops and tracking errors in their specifications (like beginning with 1 instead of 0) would be enormous were such facilities generally available.

The problem is that traditional programming languages do not attach the same meaning to program constructs as the programmer who makes use of them. For example, a programmer relates significant features of his data to the dimensions of an array and has a rich model of the structural relations embodied in the array. However, programming languages and their supporting facilities incorporate only trivial knowledge about their compound structures. In a traditional computational environment, the system's total understanding of arrays consists essentially of how to compile a subscripted reference into machine code that accesses the corresponding memory location. The way the user views array structures is quite literally "lost in translation". Consequently, programming, editing, and debugging are performed on a more primitive level than the one on which the user conceptualizes his tasks. To take a trivial example, the programmer cannot talk in terms of "the next element" or talk about a whole row of a two-dimensional array as a single object. [*APL*, due to its strong orientation towards array processing, does have some capabilities along these lines and they are much

appreciated by their users; nevertheless, their implementation is still translation oriented, and the system has no explicit model of array structures.]

Object-oriented languages such as Smalltalk take one important step in the right direction. They organize system activity around objects which understand messages, rather than in terms of operations which have objects as their arguments. As a result, information about how to print and edit structures can be stored as part of "prototype" objects. This brings some aspects of the user's model of the fundamental system structures into the system itself. Internalizing part of the user's view of these structures supports development of appropriate printing and editing facilities for them.

The user's model of the more active aspects of a computational system are much more difficult to deal with. Chapter 7 discussed the implementation of a bargraph trace for iterations. That tool was constructed primarily as an illustration of dynamic, analogic, visually presented information. It was not fully developed, in that the user had to insert code into his program that called the routine for tracing an iteration as a bargraph. Contemporary programming environments lack the means to represent the user's conception of iterative processes, recursive evaluation, or even conditional execution. This has wide-reaching ramifications. At the least, it makes it very difficult to design a monitoring system which could determine an appropriate form of tracing given only a command to trace a particular piece of code which happened to be an iteration. Moreover, there is rarely one universally correct way to print or trace a given construct. To determine a reasonable method, the system must integrate knowledge about available methods and about the particular construct with knowledge about the entire context of the computation and of the user's ongoing activity.

Given that various programming constructs have rich meaning for the programmer, there are certain basic kinds of information which the user would naturally want to obtain about computational activity involving them. Somewhat in the spirit of the schematization of system fundamentals presented in Chapter 6, the suggestion here is that a programming system which understood its own constructs would support monitoring that could provide answers to such natural questions as "Which clause of a conditional was executed?"; "What was its test? its code?"; "What is the state of the iteration?"; and "Is the recursion approaching a termination condition?". These are questions which the programmer must actually ask frequently, but without really having the vocabulary to do so. He must laboriously determine which clause of a conditional had been executed by deductively reasoning from what he can find out about the values of relevant variables in conjunction with manual simulation or machine-assisted single-stepping of the tests in the conditional. To find out how far an iteration has progressed, he must determine the name of the iterative

variable, then discover what the initial and termination values were, then compute the proportion of completion -- not always a simple task. To see if a recursion is nearing its termination, he must accumulate information about each recursive instance of the process and analyze this body of information for the patterns it contains.

The overall idea, then, is that a programming environment should incorporate a full user-oriented model of each of the language's fundamental structures and operations. The programmer should be able to simply command that a designated program construct be monitored and to obtain sophisticated printouts merely by asking to see some object. The system would then determine the important aspects of the construct, an appropriate format for communicating information, and a natural level of detail, all based on a model reflecting the user's viewpoint. The present work certainly could have pursued the development of additional individual tools and mechanisms. However, the significant barrier to the development of robust and coherent programming environments which interact with the programmer in his own terms is the lack of internal, self-aware, user-oriented models.

The generalized method of Event production discussed in Chapter 7 provides a foundation for such model-based facilities at the system level. However, as presented there, Events cannot capture the full meaning of fundamental system constructs. They isolate components of computational states or operations, but a corresponding representation of the nature of these parts, their interrelationships, and the activity occurring in the associated computation is lacking. A KRL-1 representation could have been constructed which would have incorporated some of the relevant information necessary for modelling system-level structures and operations. Yet, in addition to discouraging implementation considerations, the problem remains that even KRL-1, one of the most advanced of contemporary representation systems, is not yet rich enough to represent the *active* dependencies of the constructs. For example, a KRL-1 representation of a Process could indicate the names of the different Process fields, describe which were important from various perspectives, and say something about logical relations among the components (including temporal relations). However, it does not seem possible to successfully represent in KRL-1 the essential meaning of Processes as being active components of the system which cause computational activity, are subject to resource limitations, become suspended or resumed, and get scheduled according to some priority scheme.

Nor do other contemporary representational systems -- those based on predicate calculus or on semantic nets, for example -- seem able to accomplish much more than KRL-1 in this area. The largest barrier to pursuing the direction indicated here is the weakness of current representation languages in the domain of computational activity

and effects. [It should be noted in this regard that KRL-1 not only is unable to represent computations self-consciously, so to speak, but cannot even *specify* computations to be performed. It relies instead on the underlying implementation language (INTERLISP) for that. This is recognized as a major area for future research in the KRL effort.] Simply abstracting the relevant features for representing various kinds of computations is an inadequate approach. Computer Science and Artificial Intelligence lack even the basic conceptual framework in which the formal structures necessary for modelling computational activity could be developed. Research in this domain has produced only the most intuitive notions of how to represent computations at a high level corresponding to the user's viewpoint.

The practical and theoretical obstacles to a full modelling of system-level structures and operations not only restrict the richness and independence of monitoring facilities built according to the scheme proposed and demonstrated in this work, but it also severely limits the use of Events in the system's own computation, as suggested in Chapter 7. Moreover, the lack of a sophisticated representation for system constructs makes it impossible to develop a general method for system detection of high-level patterns in low-level phenomena, a key aspect of the principles for high-level monitoring advanced here. As things stand, any tool for the compilation of low-level information must be implemented on an *ad hoc* basis, as was done for the graphical monitoring of iterations in this work. If an extensive model of system constructs was available to the monitor, a more generalized project of automated observation and compilation could be undertaken. Even more important than improving the quality of a monitor's response to user requests are the opportunities this would provide for developing techniques investing the monitor with the ability to notice important activity and decide what to show when, rather than waiting passively for user requests. [*cf.* Chapter 4, Section D, "The Act of Debugging", where a related theme is pursued in an analysis of the nature of the computer's involvement in the different aspects of debugging.] Again, any such ambitious program would require a richer and more activity-oriented representation of computational structures and processes than currently appears feasible.

This problem is particularly important from another, more general standpoint. A central theoretical contribution which computer science has to offer (as distinct from the applications of computer technology) for the activity of other disciplines is a means of representing processes and understanding the nature of Process. This is one explanation for the attraction computer-oriented representation system hold for workers in such otherwise diverse fields as Linguistics, Psychology, Neurophysiology, Economics, and Physics. While much attention has been paid to static organizational structures and factual representations, no other discipline (including Mathematics, the usual domain of scientific language) provides any sort of language for representing (talking about, specifying, etc.) processes, nor has much been said (outside of

Philosophy) about the nature of Process itself. [Differential equations, perhaps the closest that any mathematical form comes to representing processes, actually represent the relationship between time and individual states. Thus, they deal only obliquely with the process creating that relationship. In effect, they serve as an index into an immense file of snapshots, and as such do not capture the dynamic aspects of the activity which they describe.]

The entire problem of the representation of processes and the nature of Process is therefore important in a large way, as well as having a direct bearing on the design and construction of program- and system-level monitoring facilities. Only when computational processes -- not simply computational states, operations, and sequences of these -- can be represented, both as instructions to the computational engine and within an internal, self-aware model, will system-level programming and debugging facilities be able to deal with patterns of activity, and patterns of those patterns. At that point, the computer could play a substantial role in all five phases of debugging [*cf.* Chapter 4, Section D.], instead of in monitoring only. That would permit removing even more of the monitoring burden from the user than could be accomplished following the methods of this work. This direction emerges directly from the investigations that led to the kinds of high-level monitoring facilities demonstrated in the previous chapter. It suggests an approach ripe for both theoretical investigation and successful practical implementation.

C. In Conclusion

Technological and theoretical developments in the near future should make implementation of this work's proposals generally practical, both in monitors for complex systems and in "programmer's assistants" for ordinary programming languages. The direction pointed to here is one of increased involvement of the computer in all phases of program development. More importantly, it leads to computational environments which better support human activity. By designing facilities more in accordance with the nature of human comprehension processes, programming can be made an easier, more creative, and less frustrating undertaking. In addition to the more concrete proposals it makes, this work stands as an investigation into certain aspects of intelligent behavior, both human and artificial, with an emphasis on processes of comprehension. It addresses the important theoretical challenge to understand the differences between active processes and discrete states and between symbolic and analogic information. The practical aspects of this work show how such theoretical and user-oriented phenomenological investigations can also produce technologically valuable results.

Selected Bibliography

Selected Bibliography

Anderson, John R. and Gordon H. Bower. *Human Associative Memory.* Washington: Winston, 1973.

Apple, C. T. "The Program Monitor -- A Device for Program Performance Measurement", *Proceedings of the ACM 20th National Conference*, 1965, pp. 66-75.

Arnheim, Rudolf. *Visual Thinking.* Berkeley: University of California Press, 1969.

Aron, J. D. *The Program Development Process: Part I, The Individual Programmer.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1974.

Backus, J. W., R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A Nelson, D. Sayre, P. B. Sheridan, H. Stern, L. Ziller, R. A Hughes, and R. Nutt. "The FORTRAN Automatic Coding System," *Proceedings of the Western Joint Computer Conference*, 11 (1957), 188-198. Reprinted in Rosen, *Programming Systems and Languages*, pp. 29-47.

Balzer, R. M. "EXDAMS -- EXtendable Debugging and Monitoring System. *Proceedings AFIPS Spring Joint Computer Conference*, 34 (1969), 567-580.

Barnard, David (ed.). *An Annotated Bibliography on Computer Program Engineering.* 5th ed. Computer Systems Research Group, University of Toronto, Technical Report CSRG-80, May 1977.

Barron, D. W. "Programming in Wonderland," *Computational Bulletin*, 15 (April 1971), 153.

Bashkow, Theodore, Azra Sasson, and Arnold Kronfeld. "System Design of a FORTRAN Machine," *IEEE Transactions.*, EC-16, 4 (August 1967), 485-499. Reprinted in *Computer Structures: Readings and Examples*, eds. C. Gordon Bell and Allen Newell (New York: McGraw-Hill Book Company, 1971), pp. 363-381.

Blair, Jim. "Extendable Non-Interactive Debugging," *Debugging Techniques in Large Systems*, ed. Randall Rustin, pp. 93-115.

Bobrow, Daniel G. and Bertram Raphael. *New Programming Languages for AI Research.* Xerox Palo Alto Research Center Report CSL-73-2, August 1973.

Bobrow, Daniel G. and Ben Wegbreit. "A Model for Control Structures for Artificial Intelligence Programming Languages," *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford, California, August 1973.

Bobrow, Daniel G. and Terry Winograd. *An Overview of KRL, A Knowledge Representation Language.* Xerox Palo Alto Research Center Memo CSL-76-4, July 4, 1976. Also Stanford Artificial Intelligence Laboratory Memo AIM-293 and in *Cognitive Science*, 1, 1 (1977).

Bornat, Richard and Bob Wielinga. "An Underview of KRL - Is it a Programming Language?", *AISB* (November 1976).

Brady, P. T. "Writing an Online Debugging Program for the Experienced User." *CACM* 11 (June 1968), 423-27.

Broadbent, Donald E. *Perception and Communication.* London: Pergamon Press, 1958, pp. 297-300. Reprinted in Norman, *Memory and Attention*, pp. 22-24.

Brown, A. R. and Sampson, W.A. *Program Debugging: The Prevention and Cure of Program Errors.* New York: American Elsevier, 1973.

Buchanan, Jack R. and David C. Luckham. *On Automating the Construction of Programs.* Stanford Artificial Intelligence Laboratory Memo AIM-236, May 1974.

Card, Stuart K., William K. English, and Betty Burr. *Evaluation of Mouse, Rate-Controlled Isometric Joystick, Step Keys, and Text Keys for Text Selection on a CRT.* Xerox Palo Alto Research Center Memo SSL-77-1, April 1977.

Chase, William G. (ed.). *Visual Information Processing.* New York: Academic Press, 1973.

Chase, William G. and Herbert Simon. "The Mind's Eye in Chess," *Visual Information Processing*, ed. William G. Chase, pp. 215-281.

Cheatham, Thomas. E. and Ben. Wegbreit. "A Laboratory for the Study of Automating Programming." *Proceedings AFIPS Spring Joint Computer Conference*, 40 (1972), 11-22.

Christensen, Carlos. "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language," *Interactive Systems for Experimental and Applied Mathematics*, eds. M. Klerer and J. Reinfelds. New York: Academic Press, 1968.

Christensen, Carlos. "An Introduction to AMBIT/L, a Diagrammatic Language for List Processing," *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, 1971.

Christensen, Carlos, Michael Wolfberg, and Michael Fischer. *A Report on AMBIT/G.* Wakefield, Massachusetts: Massachusetts Computer Associates, 1971.

Cofer, Charles N. (ed.). *The Structure of Human Memory.* San Francisco: W. H. Freeman and Company, 1976.

Cofer, Charles N. "A Historical Perspective, *The Structure of Human Memory*, ed. Cofer, pp. 1-14.

Crosson, Frederick. J. "Information Theory and Phenomenology," *Philosophy and Cybernetics*, ed. Crosson and Sayre, pp. 99-136.

Crosson, Frederick. J. "Memory, Models, and Meaning," *Philosophy and Cybernetics*, ed. Crosson and Sayre, pp. 193-202.

Crosson, Frederic J. and Kenneth M. Sayre (eds.). *Philosopy and Cybernetics.* New York: Clarion Books, 1968.

Davis, Randall. *Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases*, Stanford Artificial Intelligence Laboratory Memo AIM-283, July 1976 (Ph. D. Thesis).

Davis, Randall, Bruce Buchanan, and Edward H. Shortliffe. *Production Rules as a Representation for a Knowedge-Based Consultation Program.* Stanford Artificial Intelligence Laboratory Memo AIM-266, November 1975.

Dershowitz, Nachum and Zohar Manna. *The Evolution of Programs: A System for Automatic Program Modification.* Stanford Artificial Intelligence Laboratory Memo AIM-294, December 1976.

Diehm, I. C. "Computer Aids to Code Checking." *Proceedings of the ACM Toronto Meeting*, September 8-10, 1952, pp. 19-21.

Digital Equipment Corporation. *PDP-10 Reference Handbook.* Maynard, Massachusetts: Digital Equipment Corporation, 1969.

Dijkstra, Edsger W. *A Discipline of Programming.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1976.

Engelbart, D. E. and W. K. English. "A Research Center for Augmenting Human Intellect", *Proceedings AFIPS Fall Joint Computer Conference*, 33 (1968), 395-410.

English, W. K., D. C. Englebart, and M. L. Berman. "Display Selection Techniques for Text Manipulation," *IEEE Transactions on Human Factors in Electronics*, HFE-8, 1 (March 1967).

Erman, Lee D. and Victor R. Lesser. "A Multi-Level Organization for Problem Solving Using Many Diverse Cooperating Sources of Knowledge," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, USSR, 483-490.

Erman, Lee D. and Victor R. Lesser. *System Engineering Techniques for Artificial Intelligence Systems.* Carnegie-Mellon University Department of Computer Science Report, December, 1977.

Ernst, G. W. and Allen Newell. *GPS: A Case Study in Generality and Problem Solving.* New York: Academic Press, 1969.

Evans, T. G. and D. L. Darley. "DEBUG -- An Extension to Current Online Debugging Techniques. *CACM*, 8 (May 1965), 321-26.

Evans, T. G. and D. L. Darley. "On-Line Debugging Techniques: A Survey," *Proceedings AFIPS Fall Joint Computer Conference*, 29 (1966), 37-50.

Falkoff, A. D., and K. E. Iverson. "The Design of APL," *IBM Journal of Research and Development*, July 1973, 324-334.

Feldman, Jerry. A., James. R. Low, Daniel. C. Swinehart, and Russell. H. Taylor. "Recent Developments in SAIL -- An ALGOL-Based Language for Artificial Intelligence," *Proceedings AFIPS Fall Joint Computer Conference*, 41 (1972), 1193-1202.

Ferguson, H. E. and E. Berner. "Debugging Systems at the Source Language Level," *CACM*, 6 (August 1963), 430-432.

Fikes, Richard and Gary Hendrix. *A Network-Based Knowledge Representation and its Natural Deduction System.* SRI International Artificial Intelligence Center Report TN147, July 1977.

Fisher, David A. "DoD's Common Programming Language Effort," *Computer*, March 1978, 24-33.

Gaines, R. Stockton. *The Debugging of Computer Programs*, Ph.D. Thesis, Princeton University, Department of Electrical Engineering, 1969. Also Communications Research Division, Institute for Defense Analysis (Princeton, New Jersey) Working Paper No. 266, August 1969.

Galley, S. W. and R. P. Goldberg. "Software Debugging: The Virtual Machine Approach," *Proceedings of the ACM Annual Conference*, 2 (November 1974), 395-401.

Gilmartin, Kevin J., Allen Newell, and Herbert A. Simon. "A Program Modeling Short-Term Memory Under Strategy Control," *The Structure of Human Memory*, ed. Charles N. Cofer, pp. 15-30.

Goldstine, Herman H: and John von Neumann. "Planning and Coding Problems for an Electronic Computing Instrument," prepared for U. S. Army Ordnance Department, 1947-48, in John von Neumann, *Collected Works*, Volume V, *Design Of Computers, Theory of Automata, and Numerical Analysis.* Oxford: Pergamon Press, 1963, pp. 80-235.

Goldberg, Adele and Alan Kay (eds.) and The Learning Research Group. *Smalltalk-72 Instruction Manual.* Xerox Palo Alto Research Center Report SSL-76-6, March 1976.

Green, C. Cordell. "The Design of the PSI Program Synthesis System," *Proceedings of the Second International Conference on Software Engineering*, Computer Society, Institute of Electrical and Electronic Engineers, Inc., Long Beach, California, October 1976, 4-18.

Green, C. Cordell. "A Summary of the PSI Program Synthesis System," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Massachusetts, August 1977, 380-81.

Green, C. Cordell, R. J. Waldinger, David R. Barstow, Robert Elschlager, Douglas B. Lenat, Brian P. McCune, David E. Shaw, and Louis I. Steinberg. *Progress Report on Program-Understanding Systems.* Stanford Artificial Intelligence Laboratory Memo AIM-240, August 1974.

Gregory, R. L. *Eye and Brain.* New York: McGraw-Hill Book Company, 1966.

Grishman, Ralph, "Criteria for a Debugging Language," *Debugging Techniques in Large Systems*, ed. Randall Rustin, pp. 57-75.

Grishman, R. "The Debugging System AIDS." *Proceedings AFIPS Spring Joint Computer Conference*, 36 (1970), 59-63.

Halpern, M. "Computer Programming: The Debugging Epoch Opens," *Computers and Automation*, 14 (November 1965), 28-31.

Hewitt, Carl. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot.* MIT Project MAC AI Memo 251, April, 1972 (Ph. D. Thesis).

Hewitt, Carl and Brian Smith. "Towards a Programming Apprentice," *Proceedings of IEEE Transactions on Software Engineering*, 1, 1 (1975), pp. 26-45.

Hubel, D. H., and T. N. Wiesel. "Receptive Fields, Binocular Interaction, and Functional Architecture in the Cat's Visual Cortex," *Journal of Physiology*, 160, 106 (1962).

Husserl, Edmund. *Ideas: General Introduction to Pure Phenomenology*, trans. W. R. Boyce Gibson. New York: Collier Books, 1972. [The original German edition was published in 1913, and the first English edition in 1931.]

Husserl, Edmund. *The Phenomenology of Internal Time-Consciousness*, ed. Martin Heidegger, trans. James S. Churchill. Bloomington: Indiana University Press, 1964. Based on lectures during the period 1904-1910.

Ingalls, Daniel. H. *FETE: A FORTRAN Execution Time Estimator.* Stanford University Computer Science Department Report CS-71-204, February 1971.

Jonsson, S. I. "On-line Program Debugging", *BIT*, 8 (1968), 122-127.

Kintsch, Walter. *The Representation of Meaning in Memory.* Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1974.

Knuth, Donald E. *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms.* 2nd printing. Reading, Massachusetts: Addison-Wesley Publishing Company, 1969.

Koestler, Arthur. *The Act of Creation.* New York: The Macmillan Company, 1964.

Kroll, Neal E. A. "Visual Short-Term Memory," *Short-Term Memory*, eds. Diana Deutsch and J. Anthony Deutsch, pp. 154-181. New York: Academic Press, 1975.

Learning Research Group. *Personal Dynamic Media.* Xerox Palo Alto Research Center Report SSL-76-1, 1976. Excerpts published in *IEEE Computer Magazine*, March 1977.

Lesser, Victor R. and Lee C. Erman. "A Retrospective View on the Hearsay-II Architecture," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Massachusetts, 790-800.

Lettvin, J. Y., H. R. Maturana, W. S. McCulloch, and W. H. Pitts. "What the Frog's Eye Tells the Frog's Brain," *Proceedings of the Institute of Radio Engineers N. Y.*, 47, 11 (November 1959), 1940-1959,. Reprinted in *Embodiments of Mind*, Warren S. McCulloch pp. 230-255. Cambridge, Massachusetts: The MIT Press, 1965.

Lilly, John Cunningham. *The Mind of the Dolphin: A Nonhuman Intelligence.* 3rd printing. Chapter 3, "Differences between the Communication of Man and Dolphin," pp. 67-97. New York: Avon Books, 1971

Lisp Machine Group, The. *Lisp Machine Progress Report.* MIT Artificial Intelligence Laboratory Report 444, August 1977.

MacKay, Donald. M. "The Epistemological Problem for Automata," *Automata Studies*, eds. C. E. Shannon and J. McCarthy, pp. 235-252. Princeton, New Jersey: Princeton University Press, 1956.

Mann, G. A. "A Survey of Debug Systems," *The Honeywell Computer Journal*, 7, 3 (1973), 182-198.

Manna, Zohar and Richard Waldinger. *The Logic of Computer Programming.* Stanford Artificial Intelligence Laboratory Memo AIM-298, August 1977.

McCarthy, John, *et. al. LISP 1.5 Programmer's Manual.* Cambridge, Mass: The MIT Press, 1962.

Metcalfe, Robert M. and David R. Boggs. *Ethernet: Distributed Packet Switching for Local Computer Networks.* Xerox Palo Alto Research Center Report CSL-75-7, November 1975.

Miller, G. A. "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Review*, 63 (1956), 81-97.

Minsky, Marvin. "A Framework for Representing Knowledge," *The Psychology of Computer Vision*, ed. P. Winston. New York: McGraw-Hill, 1975.

Mills, H. "Top Down Programming in Large Systems," *Debugging Techniques in Large Systems*, ed. Randall Rustin. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1971.

Mitchell, James G., William Maybury, and Richard Sweet. *Mesa Language Manual.* Xerox Palo Alto Research Center Report CSL-78-1, February 1978.

Moore, J. Strother II. *The INTERLISP Virtual Machine Specification.* Xerox Palto Alto Research Center Computer Science Laboratory Report CSL 76-5, September 1976.

Moore, J. Strother II. *Text Editing Primitives -- The TXDT Package.* Xerox Palo Alto Research Center Computer Science Laboratory Internal Document, 1977.

Newell, Allen. "Production Systems: Models of Control Structures," *Visual Information Processing*, ed. William G. Chase, Chapter 10, pp. 463-526. New York: Academic Press, 1973.

Newell, Allen. "A Theoretical exploration of Mechanisms for Coding the Stimulus," *Coding Processes in Human Memory*, eds. Arthur W. Melton and Edwin Martin, pp. 373-434. Washington: V. H. Winston and Sons, 1972.

Newell, Allen and Herbert A. Simon. "GPS, A Program that Simulates Human Thought," *Lernende Automate.* Munich: R. Oldenbourg KG, 1961. Reprinted in *Computers and Thought*, eds. Edward A. Feigenbaum and Julian Feldman (New York: McGraw-Hill Book Company, 1963), pp. 279-296.

Newell, Allen, and F. M. Tonge. "An Introduction to Information Processing Language V," *CACM*, 3 (April 1960), 205-211. Reprinted in *Programming Systems and Languages*, ed. Saul Rosen, pp. 362-74.

Norman, Donald A. *Memory and Attention.* 2nd ed. New York: John Wiley and Sons, 1976.

Norman, Donald A. and Daniel G. Bobrow. "On the Role of Active Memory Processes in Perception and Cognition," *The Structure of Human Memory*, ed. Charles N. Cofer, pp. 114-132.

Norman, Donald A., David E. Rumelhart, and the LNR Research Group. *Explorations in Cognition.* San Francisco: Freeman, 1975.

Petit, Philip. *RAID.* Stanford Artificial Intelligence Laboratory Operating Note 58.1, 1970.

Pierce, J. R. *Symbols, Signals and Noise: The Nature and Process of Communication.* New York: Harper and Row, Inc., 1961.

Popper, Karl R. "Indeterminism in Quantum Physics and in Classical Physics," (in two parts), *The British Journal for the Philosophy of Science*, 1, 2 and 3 (October and November, 1950).

Raphael, Bertram, "SIR: Semantic Information Retrieval," *Semantic Information Processing*, ed. Marvin Minsky, pp. 33-145. Cambridge, Mass: The MIT Press, 1968.

Reiser, John F. *BAIL: A Debugger for SAIL.* Stanford Artificial Intelligence Laboratory Memo AIM-270, October 1975.

Rich, Charles and Howard E. Shrobe. *Initial Report on a LISP Programmer's Apprentice.* MIT Artificial Intelligence Laboratory Memo AI-TR-354, December 1976 (M. S. Thesis).

Roberts, R. Bruce and Ira P. Goldstein. *The FRL Manual.* MIT Artificial Intelligence Laboratory Memo 409, September 1977.

Roberts, R. Bruce and Ira P. Goldstein. *The FRL Primer.* MIT Artificial Intelligence Laboratory Memo 408, July 1977.

Rosen, Saul (ed.). *Programming Systems and Languages.* New York: McGraw-Hill Book Company, 1967.

Rulifson, J. F. *et al. QA4: A Procedural Calculus for Intuitive Reasoning.* SRI AI Center Technical Note 73, November 1973.

Rustin, Randall (ed.). *Debugging Techniques in Large Systems.* Courant Computer Science Symposium 1, June 29 -- July 1, 1970. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1972.

Satterthwaite, Edwin H., Jr. *Source Language Debugging Tools.* Stanford Computer Science Department Report CS-75-494, May 1975 (Ph. D. Thesis).

Satterthwaite, E. "Debugging Tools for High-Level Languages," *Software -- Practice and Experience* 2, 3 (July-September 1975), 197-217.

Schwartz, Jacob T. "An Overview of Bugs," *Debugging Techniques in Large Systems,* ed. Randall Rustin, pp. 1-16.

Seidel, Kenneth P. "Debugging Past and Present," *Software Age,* August 1968, 22-28.

Shortliffe, Edward H. *MYCIN -- A Rule-Based Computer Program for Advising Physicians Regarding Antimicrobial Therapy Selection.* Stanford Artificial Intelligence Memo AIM-251, October 1974 (Ph. D. Thesis). Also published as *MYCIN: Computer-based Medical Consultations*; New York: American Elsevier, 1976.

Simon, Herbert A. *The Sciences of the Artificial.* Cambridge, Massachusetts: The MIT Press, 1969.

Smith, David C. *Pygmalion: A Creative Programming Environment,* Stanford Artificial Intelligence Laboratory Memo AIM-260, June 1975 (Ph. D. Thesis).

Sproull, Robert F. *Interlisp Display Primitives.* Xerox Palo Alto Research Center Computer Science Laboratory Internal Document, July 1977.

Sproull, Robert F. and E. L. Thomas. "A Network Graphics Protocol," *Computer Graphics, SIGGRAPH Quarterly,* Fall 1974.

Supnik, Robert M. "Debugging Under Simulation," *Debugging Techniques in Large Systems,* ed. Randall Rustin, pp. 117-135.

Sussman, Gerald Jay. *A Computational Model of Skill Acquisition.* MIT Artificial Intelligence Laboratory Memo TR-297, Cambridge, Mass, August 1973 (Ph. D. Thesis).

Sussman, Gerald Jay and Drew Vincent McDermott. *Why Conniving is Better than Planning.* MIT Project MAC AI Memo 255A, April 1972.

Sussman, Gerald Jay and Terry Winograd. *MICRO-PLANNER Reference Manual.* MIT Project MAC AI Memo 203, July 1970.

Sutherland, Ivan C. *SKETCHPAD: A Man-Machine Graphical Communication System.* MIT Lincoln Laboratory Memo TR 296, May 1965.

Swinehart, Daniel C. *COPILOT: A Multiple Process Approach to Interactive Programming Systems.* Stanford Artificial Intelligence Laboratory Memo AIM-230, July 1974 (Ph. D. Thesis).

Swinehart, Daniel C. and Robert F. Sproull. *SAIL.* Stanford Artificial Intelligence Laboratory Operating Note 57.2, January 1971.

Teitelman, Warren. "Automated Programmering -- The Programmer's Assistant," *Proceedings AFIPS Fall Joint Computer Conference,* 41 (1972), 917-921.

Teitelman, Warren. "CLISP -- Conversational LISP," *Proceedings of the Third International Joint Conference on Artificial Intelligence,* Stanford, California, August 1973, 686-690.

Teitelman, Warren. *A Display Oriented Programmer's Assistant.* Xerox Palo Alto Research Center Report CSL 77-3, March 8, 1977. A somewhat abbreviated form of this report is to be found in the *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* Cambridge, Mass, August 1977, 905-915.

Teitelman, Warren. *INTERLISP Reference Manual.* 2nd revision. Palo Alto, CA: Xerox Palo Alto Research Center, December 1975. 3rd revision, October 1978.

Teitelman, Warren. "Toward a Programming Laboratory," *Proceedings of the First International Joint Conference on Artificial Intelligence,* 1969.

TENEX JSYS Manual: A Manual of TENEX Monitor Calls. 2nd revision. Bolt, Beranek and Newman, Inc., September 1973.

von Host, Eduard. "Relations between the Central Nervous System and the Peripheral Organs," *Animal Behavior,* 2 (1954), 89-94. Reprinted in *Perception: Selected Readings in Science and Phenomenology,* ed. Paul Tibbetts (New York: Quadrangle Books, 1969), pp. 106-116.

Wanner, Eric. *On Remembering, Forgetting, and Understanding Sentences: A Study of the Deep Structure Hypothesis (Janua Linguarum, Series Minor,* 170). The Hague: Mouton, 1974.

Waugh, N. C. and Donald A. Norman. "Primary Memory," *Psychological Review,* 72 (1965), 89-104.

Winograd, Terry. "Breaking the Complexity Barrier (Again)," *Proceedings of the ACM SIGIR-SIGPLAN Interface Meeting,* November 1973.

Winograd, Terry. "Computer Memories: A Metaphor for Memory Organization," ed. Charles N. Cofer, *The Structure of Human Memory,* pp. 133-161.

Winograd, Terry. *Five Lectures on Artificial Intelligence.* Stanford Artificial Intelligence Laboratory Memo AIM-246, September 1974.

Woods, William A. "What's in a Link: Foundations for Semantic Networks," *Representation and Understanding: Studies in Cognitive Science*, eds. Daniel G. Bobrow and Allan Collins, pp. 35-82. New York: Academic Press, Inc., 1975.

Wooldridge, Dean E. *The Machinery of the Brain.* Chapter 3, "Peripheral Data Processing in the Nervous System," pp. 37-54. New York: McGraw-Hill Book Company, Inc., 1963.

Wooldridge, Dean E. *Mechanical Man: The Physical Basis of Intelligent Life.* Chapter 4, "Dandelions and Men," pp. 36-47. New York: McGraw-Hill Book Company, Inc., 1968.

Yarwood, Edward. *Toward Program Illustration.* University of Toronto Computer Systems Research Group Technical Report CSRG-84, October 1977 (M. Sc. Thesis).

Zimmerman, L. L. "On-line Programming Debugging -- A Graphic Approach," *Computers and Automation*, November 1967, 30-34.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

XEROX® is a trademark of XEROX CORPORATION Printed in U.S.A.